# THE APPLICATION OF MICROPROCESSORS AND OTHER LSI DEVICES TO DIGITAL SIGNAL PROCESSING

J. W. Gault and R. W. Stroh
Electrical Engineering Department
North Carolina State University
Raleigh, NC 27607

31 January 1976



U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35809

*Prepared for:*

Advanced Sensors Directorate
US Army Missile Research, Development
and Engineering Laboratory
US Army Missile Command
Redstone Arsenal, Alabama 35809

## DISPOSITION INSTRUCTIONS

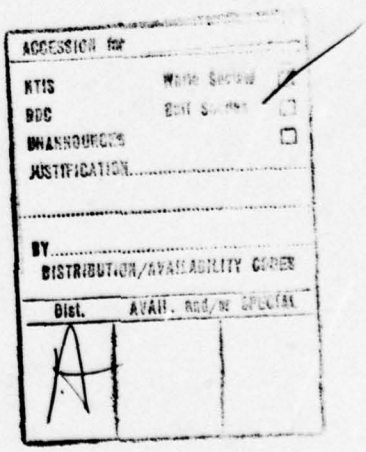
DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.

## DISCLAIMER

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

## TRADE NAMES

USE OF TRADE NAMES OR MANUFACTURERS IN THIS REPORT DOES NOT CONSTITUTE AN OFFICIAL INDORSEMENT OR APPROVAL OF THE USE OF SUCH COMMERCIAL HARDWARE OR SOFTWARE.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) (6) The Application of Microprocessors and Other LSI Devices to Digital Signal Processing, | | 5. TYPE OF REPORT & PERIOD COVERED |
| | | 6. PERFORMING ORG. REPORT NUMBER Battelle Task 72-215 |
| 7. AUTHOR(s) (10) J. W. Gault & R. W. Stroh<br>North Carolina State University<br>D. W. Burlage (COTR)<br>U.S. Army Missile Command | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Electrical Engineering Department<br>North Carolina State University<br>Raleigh, N.C. 27607 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>1M362303 A214<br>AMCMS 632303.11 21401 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Commander, US Army Missile Command<br>Attn: DRSMI-RPR<br>Redstone Arsenal, AL 35809 | | (11) 12. REPORT DATE 31 Jan[illegible]976<br>13. NUMBER OF PAGES 61 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 67p. | | 15. SECURITY CLASS. (of this report) UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(16) 1M362303A214

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Radar
Signal Processing
Microprocessors
Programmable Logic Array

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A study was performed to determine the feasibility of implementing current and future radar signal processors of Army missile systems with microprocessor hardware. Parameters such as speed, power, weight, cost and logic complexity were estimated for microprocessors implementations of these systems. Moreover, detail designs were performed for vector magnitude and monopulse angle error units of a radar processor. Comparative designs were obtained using Intel 8080 and 3000 systems, programmable logic arrays, and conventional logic.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

404 862

# The Application of Microprocessors and Other LSI Devices to Digital Signal Processing

by

J. W. Gault

R. W. Stroh

Electrical Engineering Department
North Carolina State University
Raleigh, NC 27607

31 January 1976

Final Report for Battelle Task 75-215

Submitted to:

The Army Missile Command
Redstone Aresnal, AL 35809

ATTN: Dr. D. Burlage, AMSMI-REG, Building 5400

# The Application of Microprocessors and other LSI Devices to Digital Signal Processing

## OUTLINE

i. General Objective
ii. Abstract

1. Introduction

   1.1. The intent of the study
   1.2. The evolution of specific goals
   1.3. The approach to be taken
   1.4. The organization of the paper

2. An Overview

   2.1. Microprocessor performance

      1. timing and throughput demands
      2. a bound for microprocessor performance

   2.2. Improving performance
   2.3. A general module

3. The RMS Module (A Case Study)

   3.1. The Intel 3000 implementation
   3.2. A programmable logic binary (PLA) implementation
   3.3. Comparison of approaches to the existing conventional logic approach

4. The Angle Tracking Computer (ATRAC) (A Case Study)

   4.1. The existing system
   4.2. Projected expansion of system capability
   4.3. An Intel 8080 implementation
   4.4. An Intel 3000 implementation
   4.5. Using a PLA
   4.6. Comparison of approaches

5. Summary, Conclusions and Recommendations

General Task Objectives:

1. To study current and future signal processing requirements of Army Missile Command on-board missile guidance systems and ground-based search, acquisition and tracking systems.

2. To estimate parameters such as speed, power, weight, cost and logic complexity for microprocessor implementations of these systems.

The following report describes the investigation and recommendations made with respect to the task defined above.

## Abstract

The Experimental Array Radar (EAR) used by the Army Missile Command (MICOM) was taken as a specific system against which a study could be directed.

The overall EAR system was studied with the idea of replacing conventional logic with microprocessors in mind. The performance of an Intel 8080 was bench marked against the system requirements.

Effort was then focussed on a modular organization of the EAR processor. The Root Mean Square (RMS) module was taken as a case study and implemented with an 8080, an Intel 3000 and a Programmable Logic Array (PLA). These approaches were compared to the existing conventional logic implementation.

The Angle Tracking Computer (ATRAC) was then taken as a more extensive example. Similar approaches were taken to ATRAC and compared.

The experience of this investigation is summarized and projections and recommendations are made.

# 1. INTRODUCTION

The last few years has been a period of rapid change in the area of digital electronics. One of the major advances has been the widespread availability of microprocessor devices. The devices have many attributes which make them attractive to both ground-based experimental radar and airborne guidance systems. This report is directed toward applying LSI devices to digital signal processing tasks.

## 1.1. The Intent of the Study

Basically the objective of this study was to determine the suitability of currently available devices to present and projected systems within the Army Missile Command (MICOM) at Redstone Arsenal, Alabama. In addition, projections were to be made concerning the trends in the technology and to make recommendations concerning future applications.

## 1.2. Specific Goals

Beginning with this fairly general charter more specific goals were set. The initial focus was on ground-based rather than airborne systems. This is motivated by the availability of documentation and the complexity of the problem. The problems associated with ground equipment are less restrictive and must be understood before airborne systems can be tackled.

The Experimental Array Radar (EAR) system was selected as a case study for a number of reasons. It offers a good opportunity to study the problems at hand and it is well documented. There is a good potential payoff to lab personnel in that the ideas present may be tried in a fairly easy way. The flexibility and changeability of microprocessor implementations make them very attractive to an experimental environment.

The goals of this study were made specific with the above reasoning in mind. The specific goals are given below.

a) Bound the throughput potential of a typical microprocessor,

b) Look at various implementation approaches in detail for a specific digital signal processing module,

c) Compare the performance of the various approaches,

d) Generalize the ideas and concepts gleaned from the specific examples, and

e) Make recommendations based on the study results.

## 1.3. Specific Approach

The Intel 8080 and 3000 systems were selected as the devices to be used for the detailed study. These two families are representative of very large classes of available devices and are likely to be available for the immediate future. Much of the details of these machines can be generalized to other manufacturers' devices.

It was deemed desirable to take a few problems and work on them in detail rather than to deal in the somewhat vague conclusions possible with an overall general approach. The learnings gained from the detailed examples will be generalized as much as possible.

The Root Mean Square (RMS) module of the digital signal processor portion of the EAR system was selected. This module is analyzed from a number of different approaches.

The Angle Tracking Computer (ATRAC) within the same system was selected as a somewhat more extensive example. A similar approach was taken to analyze various approaches to implementing it.

The outcome of examining these should give a reasonable indication of the suitability of the basic approach.

## 1.4. The Organization of the Report

In the following section some general considerations are given to set the stage for the examples. Sections three and four then present the

details of the two case studies (the RMS module and ATRAC). The section outlines the conclusions derived from the study and the recommendations for future planning are given.

## 2. AN OVERVIEW

The basic intent of this section is to set the framework for the detailed examples which will follow. The basic design objectives motivating the use of microprocessors are

a) the reduced component count,

b) a programmable approach to logic design, and

c) adaptable implementations.

The following subsections deal with the basic nature of microprocessors and the problem organization approach taken in their utilization.

### 2.1. Microprocessor Performance

A basic microcomputer system configuration is given in Figure 2.1. This system is typical of what is required to solve the types of problems being considered.

The design problem basically reduces to interfacing the problem to the microcomputer and writing the appropriate programs. Typical input/output ports and the signal lines are given in Figures 2.2, 2.3 and 2.4.

One of the fundamental concerns in utilizing this type of system is the system throughput. That is to say, how fast can the system accept inputs, perform a computation and produce the desired output.

Many applications will not have a critical requirement with respect to throughput. On the other hand, the speed requirements in the problems of signal processing are the essential design requirement.

It is interesting to look at throughput in a general way. By doing this the limitations on timing can be bounded. For example, the general requirement is to

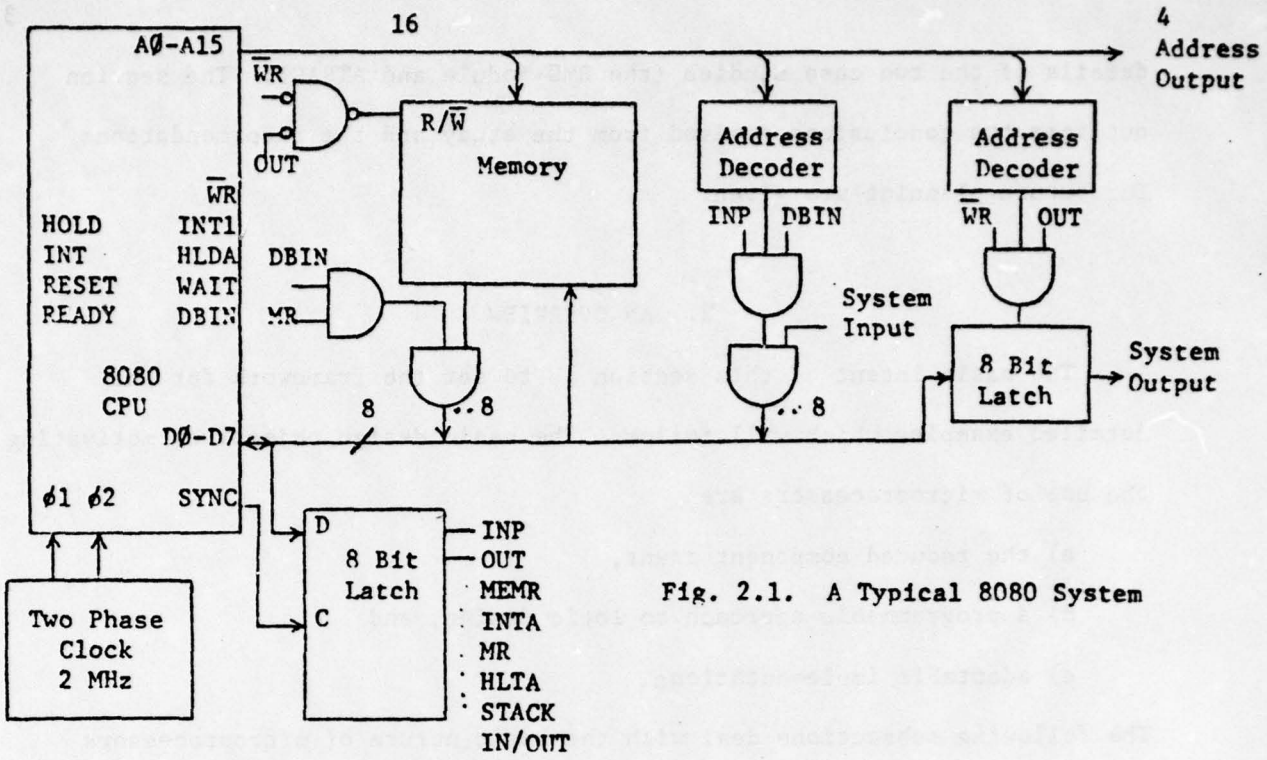



Fig. 2.1. A Typical 8080 System

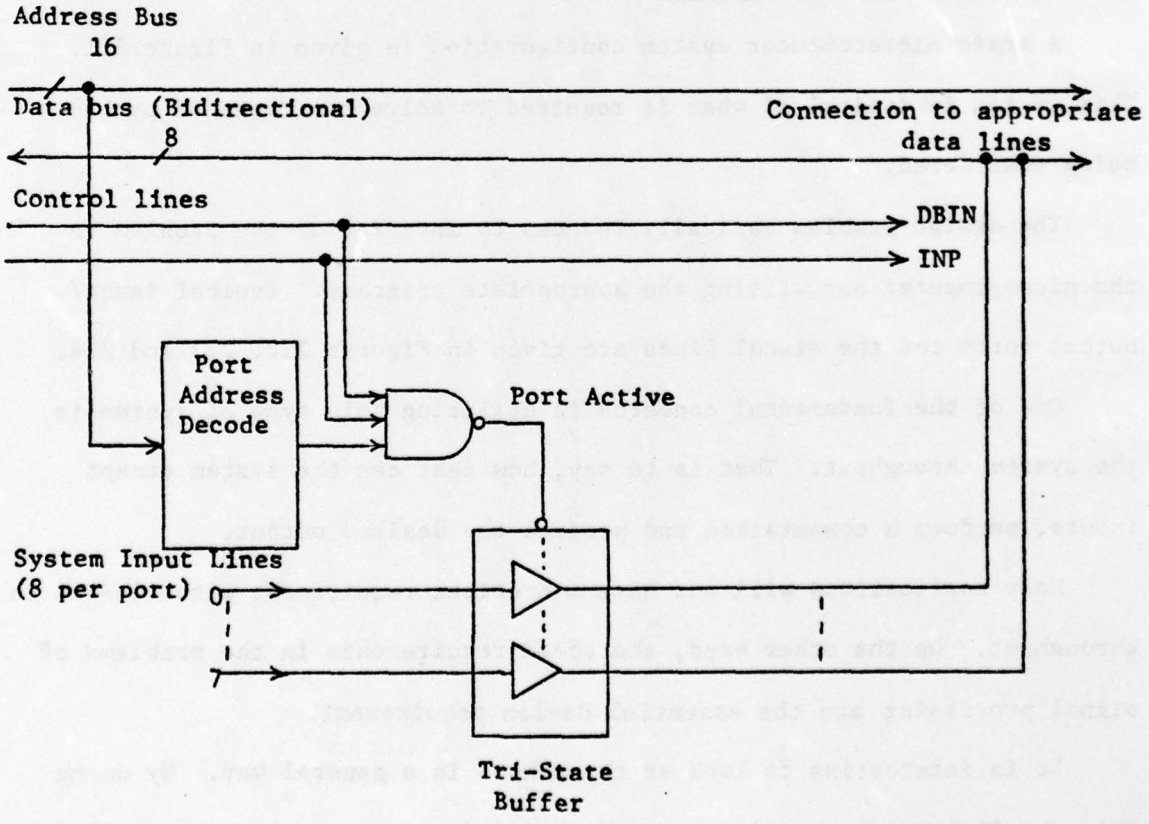



Fig. 2.2. Typical Input Port

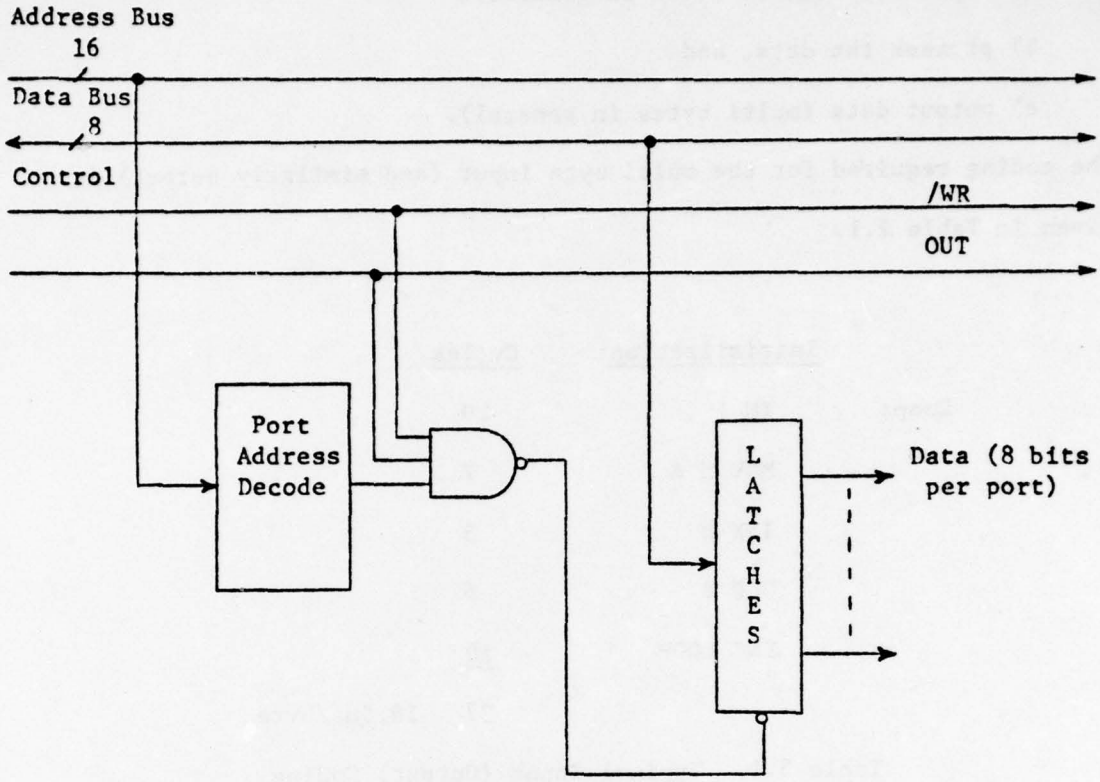



Fig. 2.3. Typical Output Port

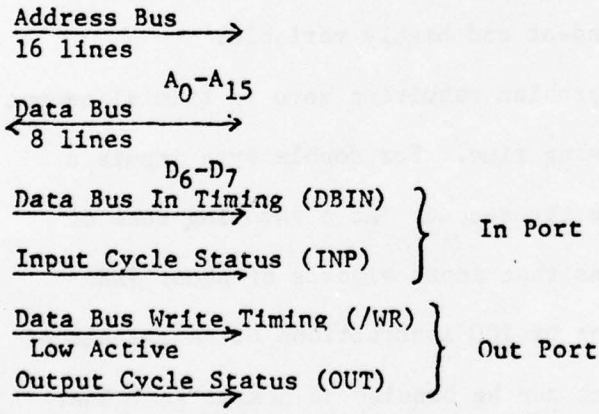



Fig. 2.4. Port Interface Lines

a) input data (multi bytes in general),

b) process the data, and

c) output data (multi bytes in general).

The coding required for the multi byte input (and similarly output) is given in Table 2.1.

| | Initialization | Cycles | |
|---|---|---|---|
| Loop: | IN 1 | 10 | |
| | MOV M,A | 7 | |
| | INX H | 5 | |
| | DCR B | 5 | |
| | JNZ LOOP | 10 | |
| | | 37 | 18.5μs/byte |

Table 2.1. Typical Input (Output) Coding

It can be seen then, that the basic overhead to input and output data is about 37 μs per byte. Now, how about the required processing time? This is of course very problem dependent and highly variable.

Consider, as a lower bound, a problem requiring zero (a trivial amount of time with respect to I/O) processing time. For double byte inputs a total of $13.5*10^3$ passes can be made per second. At a sampling rate of twice the signal frequency this means that input signals of about 7KHz can be handled. For processing steps of 100 instructions at an average of 4ms per instruction, the signal which can be handled is nearer to 700hz.

An 8x8 multiply (Table 2.2) requires about 370ms of processing time so it is clear that 400μs of total processing is potentially a very low estimate for many applications. This gives an indication of the speed limitations of single microcomputer systems with 8080-like capabilities.

```
Label   Code   Operand          Comment
MADD:   LXI    B,FIRST  ; B and C address FIRST
        LXI    H,SECND  ; H and L address SECND
        XRA    A        ; Clear carry bit
LOOP:   LDAX   B        ; Load byte of FIRST
        ADC    M        ; Add byte of SECND
                        ; with carry
        STAX   B        ; Store result at FIRST
        DCR             ; Done if  = 0
        JZ     DONE
        INX    B        ; Point to next byte of
                        ; FIRST
        INX    H        ; Point to next byte of
                        ; SECND
        JMP    LOOP     ; Add next two bytes
DONE:   ---
        ---
FIRST:  DB     90H
        DB     0BAH
        DB     84H
SECND:  DB     8AH
        DB     0AFH
        DB     32H
```

```
MULT:   MVI    B, 0  ; Initialize most significant byte
                     ; of result
        MVI    E, 9  ; Bit counter
MULT0:  MOV    A, C  ; Rotate least significant bit of
        RAR          ; multiplier to carry and shift
        MOV    C, A  ; low-order byte of result
        DCR    E
        JZ     DONE  ; Exit if complete
        MOV    A, B
        JNC    MULT1
        ADD    D     ; Add multiplicand to high-
                     ; order byte of result if bit
                     ; was a one
MULT1:  RAR          ; Carry=0 here; shift high-
                     ; order byte of result
        MOV    B, A
        JMP    MULT0
DONE:
```

```
DIV:    MVI    E, 9     ; Bit counter
        MOV    A, B
DIV0:   MOV    B, A
        MOV    A, C     ; Rotate carry into C register;
        RAL             ; rotate next most significant
                        ; bit to carry
        MOV    C, A
        DCR    E
        JZ     DIV1
        MOV    A, B     ; Rotate most significant bit to
        RAL             ; high-order quotient
        SUB    D        ; Subtract divisor. If less than
        JNC    DIV0     ; high-order quotient, go to
                        ; DIV0
        ADD    D        ; Otherwise add it back
        JMP    DIV0
DIV1:   RAL
        MOV    E, A
        MVI    A, 0FFH  ; Complement the quotient
        XRA    C
        MOV    C, A
        MOV    A, E
        RAR
DONE:
```

Table 2.2. Typical Multibyte Operations

In the EAR system the most restrictive time demand is for a throughput of approximately 200ns. These figures make it clear that an 8080-like system cannot be used to directly replace the digital signal processor.

## 2.2. Improving Performance

There are many possible approaches which may be taken to improve the speed performance of a microcomputer system. These basically fall into two categories:

a) modification of the microprocessor system itself, and

b) reorganization of the problem at a higher level.

Modification of the microcomputer system involves a tradeoff of software for hardware. Improvements may be gained from

a) utilizing direct memory access (DMA) to increase input/output speed,

b) the use of interrupts to reduce polling time, and

c) the introduction of faster logic operations for critical function (such as multiply).

These ideas will not be pursued here due to the extremely large speed increase required. It seems more fruitful to look for improvements by reorganizing the system approach.

System organization schemes for increasing speed which are well documented in the literature are

a) parallel operations,

b) pipeline operations, and

c) memory management

cache
overlap

The key to applying these ideas are the identification of system modules and their interconnection. For the purposes of this report the existing modularization of the signal processor will be used (Figure 2.5).

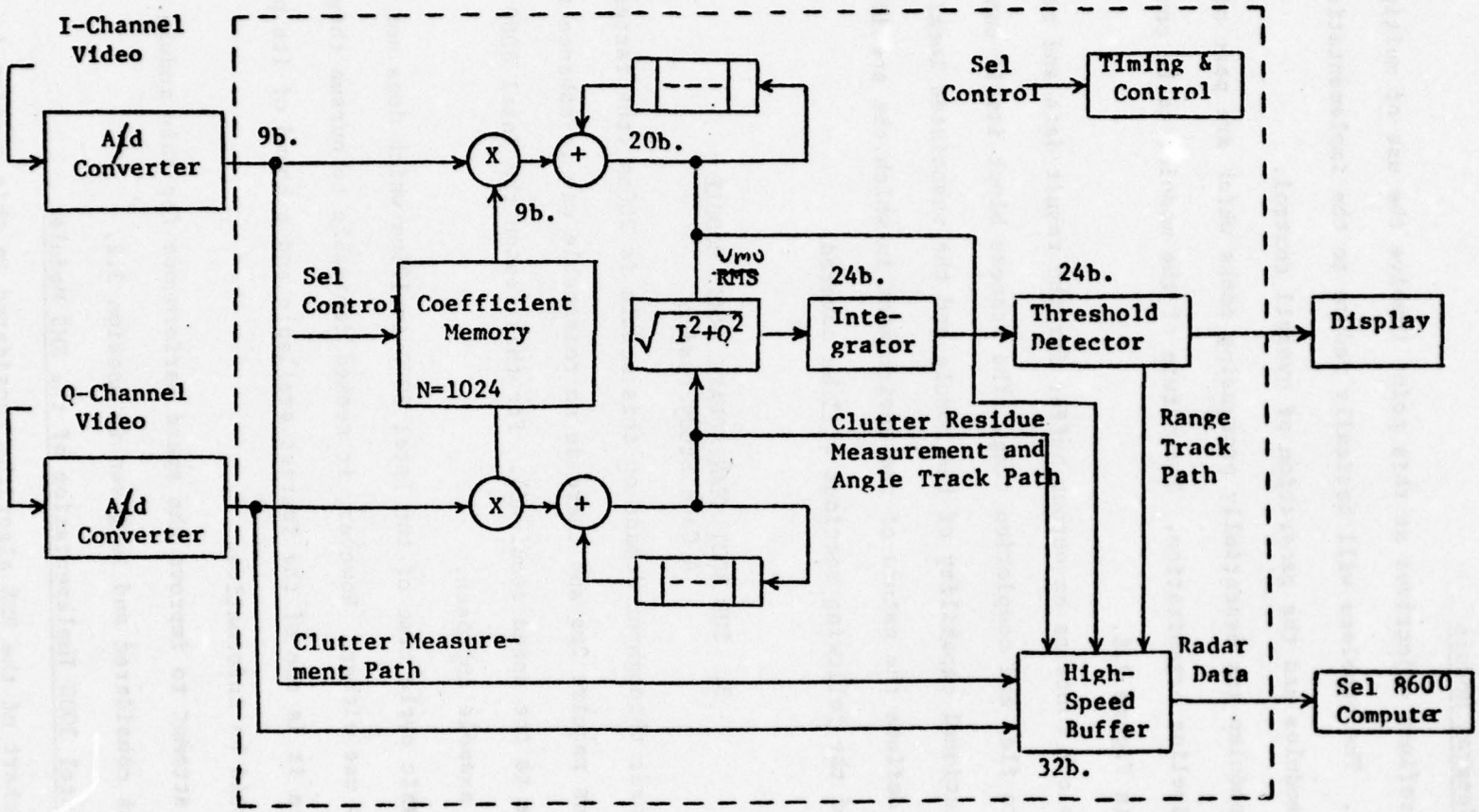



Fig. 2.5. Block Diagram of Current Programmable Signal Processor

## 2.3. A General Module

The refined objectives at this point involve the use of multiple microcomputers. The problems will basically reduce to the implementation of specific modules and the generation of overall control.

The modules are essentially processing tasks which are part of an overall pipeline organization. The nature of the modules to be considered is given in Figure 2.6.

The block contains an output buffer for the result data and two flags, an initiate flag and completion flag. The process block itself contains the computational capability of the module and the associated local control.

This defines the nature of the environment in which the specific examples of the following sections will be covered.

# 3. THE ROOT MEAN SQUARE (RMS) MODULE
## A Case Study Design

The basic throughput demand on this module is 200ns (the fastest instructions require 2μs and there is no reasonable way to enhance the performance to the speed required). For this reason the Intel 8080 was not used as an example approach.

The basic cycle time of the Intel 3000 is 125ns which does not encourage its use either. However, it seemed desirable to pursue this device since it is one of the fastest available and a bound of its performance would be informative.

In an attempt to improve the speed performance for this module a PLA approach was considered and is given in section 3.2.

## 3.1. The Intel 3000 Implementation of the RMS Module

A flowchart of the RMS algorithm considered in this section is shown in Figure 3.1. This algorithm approximates $I^2+Q^2$ to within 2% without requiring computation of a square root. It does require the computational

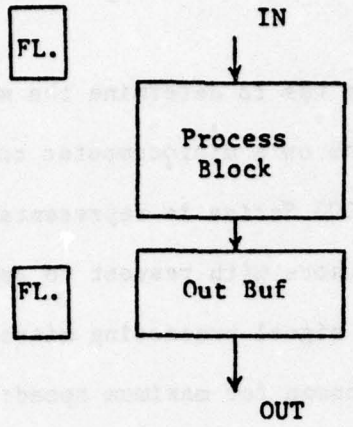



Fig. 2.6. Buffered Process Block

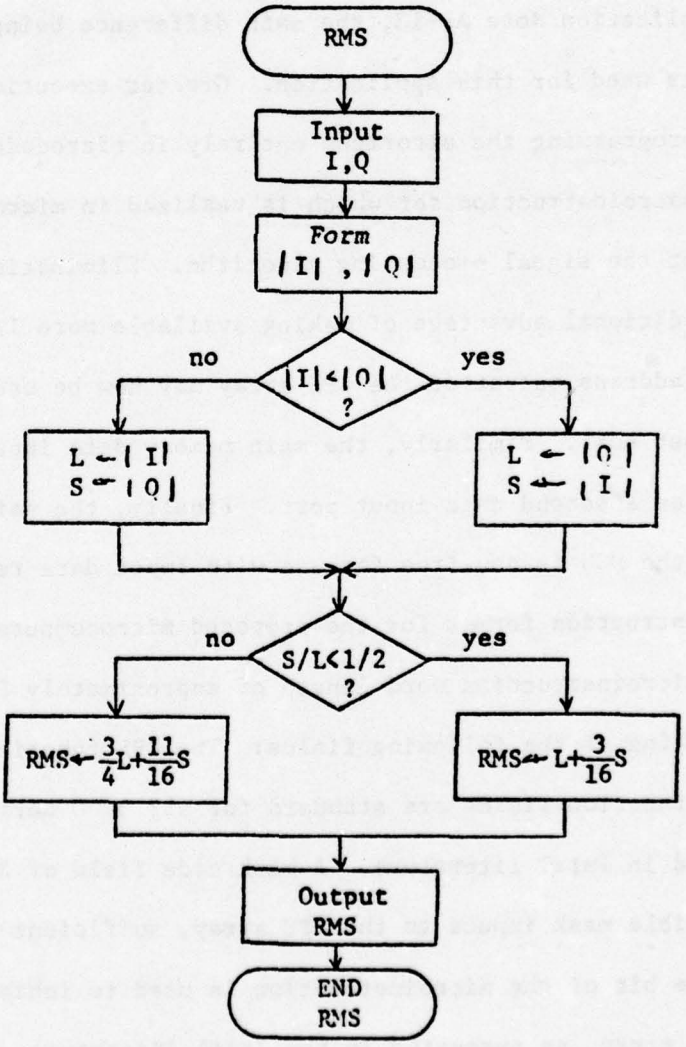



Fig. 3.1. RMS Algorithm (<2% Error)

operations of absolute value, comparison, multiplication by simple fractions and addition.

The aim of this case study was to determine the minimum computation time required for this algorithm on a microcomputer constructed of Intel 3000 Series components. The 3000 Series is representative of the current state-of-the-art in microprocessors with respect to speed. Figure 3.2 shows the configuration of the signal processing microcomputer used here. A pipelined organization was chosen for maximum speed; by overlapping microinstruction fetch and execute cycles, a microinstruction time of 125ns* is possible. The system shown in Figure 3.2 is similar to that in Figure 2 of the Intel Application Note AP-13, the main difference being that no main program memory is used for this application. Greater execution efficiency is possible by programming the algorithm entirely in microcode, as opposed to designing a macroinstruction set which is realized in microcode and then used to implement the signal processing algorithm. Elimination of main memory has an additional advantage of making available more I/O ports. The main memory address output of the CPE array may now be used as a second data output port. Similarly, the main memory data input to the CPE may now be used as a second data input port. Finally, the main memory opcode input to the MCU is now free for use with input data ready flags, etc.

The microinstruction format for the proposed microcomputer is shown in Figure 3.3. A microinstruction word length of approximately 23 bits is required, consisting of the following fields: The CPE function, flag logic, and jump function fields are standard for all 3000 Series systems and are described in Intel literature. A mask code field of 2 bits allows one of four possible mask inputs to the CPE array, sufficient for this application. One bit of the microinstruction is used to inhibit the clock input to the CPE array, as suggested in the Intel literature. This allows

*Recently this has been improved by about 30%.

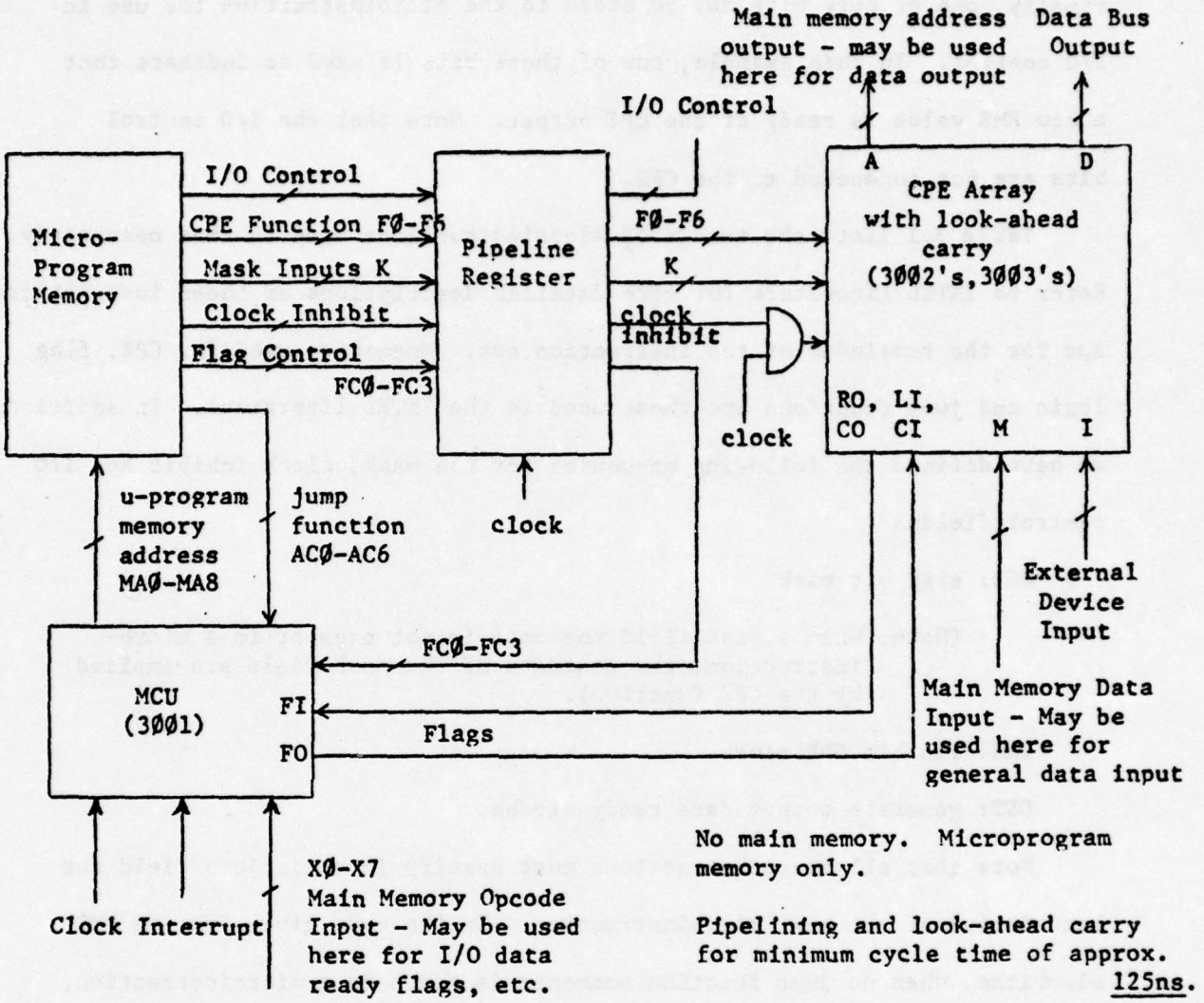



Fig. 3.2. A Signal Processing Computer Using INTEL 3ØØØ Series Components

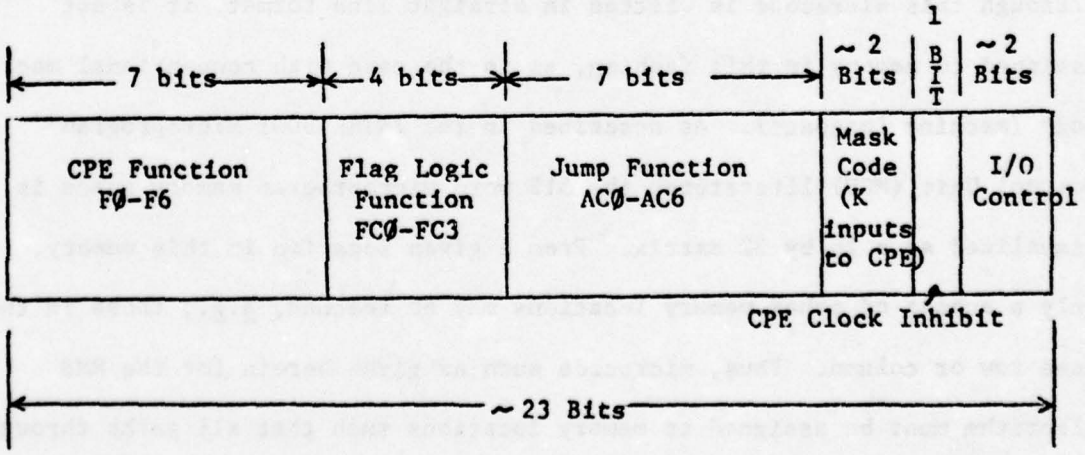



Fig. 3.3. Microinstruction Format for INTEL 3ØØØ-Based Signal Processor

the testing of sign bits without changing the contents of any CPE registers. Finally, one or more bits may be added to the microinstruction for use in I/O control. In this example, one of these bits is used to indicate that a new RMS value is ready at the CPE output. Note that the I/O control bits are not connected to the CPE.

Table 3.1 lists the subset of microinstructions used in this case study. Refer to INTEL literature for more detailed descriptions of these instructions and for the remainder of the instruction set. Mnemonics used for CPE, flag logic and jump functions are those used in the INTEL literature. In addition, we have defined the following mnemonics for the mask, clock inhibit and I/O control fields:

KSB: sign bit mask

(Note: When a mask field mnemonic is not present in a microinstruction, the contents of the mask field are implied by the CPE function).

INH: inhibit CPE clock

OST: generate output data ready strobe.

Note that all microinstructions must specify in their jump field the location(s) of the next microinstruction. In the code given for the RMS algorithm, when no jump function mnemonic is given in a microinstruction, a jump to the next instruction in the microprogram is implied. However, although this microcode is written in straight line format, it is not assigned to memory in this fashion, as is the case with conventional macrocode (machine language). As described in the INTEL 3001 Microprogram Control Unit (MCU) literature, the 512 word microprogram memory space is visualized as a 16 by 32 matrix. From a given location in this memory, only a subset of other memory locations may be reached, e.g., those in the same row or column. Thus, microcode such as given herein for the RMS algorithm must be assigned to memory locations such that all paths through

Table 3.1. The Subset of INTEL 3000 Microinstructions Used to Program the RMS Algorithm.

Symbols

$R_n$ : Any of the registers R0 thru R9, T, or AC
I : External device input data
M : Main memory data - used here for general data input
MAR : Main memory address register - used here for auxiliary data output.

| Transfer Instructions: | Function |
|---|---|
| ILR($R_n$) FF0 | $R_n \rightarrow$ AC |
| ACM(AC or T) FF0 | M → AC or T |
| LMI($R_n$) FF0 | $R_n \rightarrow$ MAR |
| SDR($R_n$) FF1 | AC → $R_n$ |
| LDI(AC or T) FF1 | I → AC or T |
| **Add Instructions:** | |
| ALR($R_n$) FF0 | AC + $R_n \rightarrow R_n$, AC |
| ADR($R_n$) FF0 | AC + $R_n \rightarrow R_n$ |
| **Complement Instruction:** | |
| CIA(AC or T) FF1 | Two's complement of AC or T → AC or T |
| **Shift Instruction:** | |
| SRA(AC or T) FF0 | Shift AC or T right one bit — 0 replaces MSB |
| **Sign Testing Instruction:** | |
| TZR($R_n$) FF0 KSB INH | Sign bit of $R_n$ → MCU flag input |
| **MISC:** | |
| NOP | No operation — sometimes required due to pipelining |
| **JUMPS:** (These are combined with one of the above CPE functions) | |
| JFL (A,B) | Take next microinstruction from location A or B depending upon CPE carry output. |
| JCF (A,B) | Take next microinstruction from location A or B depending upon state of C-flag (see STC below) |
| JPX (A,B) | Take next microinstruction from location A or B depending upon state of one of the MCU inputs X4-X7 |
| JMP (A) | Take next microinstruction from location A. This is a pseudo-instruction representing one of several unconditional jumps, e.g., |

Table 3.1 (Continued):

| | Function |
|---|---|
| JMP (A) (cont'd.) | JCC - Jump in current column,<br>JZR - Jump to row zero.<br>The particular one used depends upon the assignment of the code to memory. |

Note: All microinstructions must contain a jump code indicating the location of the next instruction.

SET FLAG CODE: (This is combined with a CPE function code)

| | |
|---|---|
| STC | Set C-flag to CPE carry output. |

---

the code are possible and the appropriate jump function code added to each instruction. This task can be more time-consuming than writing the code itself, particularly if the total number of microinstructions approaches 512, the size of the control memory. These constraints on memory assignment represent a major disadvantage of the INTEL 3000 Series: difficulty of microprogramming. The assignment of the RMS algorithm microcode to memory was not attempted due to time limitations on this study.

The microcode for the implementation of the RMS algorithm of Fig. 3.1 on the INTEL 3000 pipelined signal processing microcomputer of Fig. 3.2 is listed below. The I and Q data inputs are assumed to be in two's complement form and to be connected to the CPE's M and I input ports, respectively. An input data ready flag appears on one of the MCU inputs X4-X7. The RMS value is output from the CPE's MAR port, and an output data ready signal is provided from microprogram memory through the pipeline register. Note that because of the pipeline organization of the processor, conditional jumps on a flag (sign bit) cannot appear in the same microinstruction as the generation of the flag. Note also that multiplications are done by the conventional shift-and-add method, but the number of microinstructions is relatively small due to the constant multipliers being simple binary fractions.

```
START:  ACM(AC) FFØ                     /*GET I FROM M BUS INTO AC*/
        TZR(AC) FFØ KSB INH             /*TEST SIGN BIT OF I*/
        LDI(T) FF1 JFL(IP,IN)           /*GET Q FROM I BUS INTO T*/
  IN:   CIA(AC) FF1                     /*IF I IS NEGATIVE, COMPLEMENT IT*/
  IP:   TZR(T) FFØ KSB INH              /*TEST SIGN BIT OF Q*/
        SDR(Ø) FF1 JFL(QP,QN)           /*STORE |I| IN RØ*/
  QN:   CIA(T) FF1                      /*IF Q IS NEGATIVE, COMPLEMENT IT*/
  QP:   ILR(T) FFØ                      /*GET |Q| IN AC*/
        SDR(1) FF1                      /*AND SAVE IN R1*/
        CIA(AC) FF1                     /*FORM - |Q| IN AC*/
        ADR(Ø) FFØ STC INH              /*ADD |I| & SAVE ONLY SIGN BIT OF
                                          |I|-|Q| IN C-FLAG*/
        NOP JFL(LI,LQ)                  /*GET L IN AC*/
  LI:   ILR(Ø) FFØ JMP(LD2)
  LQ:   ILR(1) FFØ
 LD2:   SRA(AC) FFØ                     /*DIVIDE L BY 2*/
        SDR(T) FF1 JCF(SQ,SI)           /*SAVE L/2 IN T*/
  SQ:   ILR(1) FFØ JMP(CSL)             /*GET S IN AC*/
  SL:   ILR(Ø) FFØ
 CSL:   CIA(AC) FF1                     /*FORM -S IN AC*/
        ADR(T) FFØ INH                  /*TEST SIGN OF L/2-S*/
        CIA(AC) FF1 JFL(LT,GT)          /*GET +S IN AC*/
   LT:  SRA(AC) FFØ                     /*S/L < 1/2*/
        SRA(AC) FFØ                     /*DIVIDE S BY 8*/
        SRA(AC) FFØ
        SDR(T)  FF1                     /*STORE S/8 in T*/
        SRA(AC) FFØ                     /*FORM S/16 IN AC*/
        ADR(T)  FFØ JCF(LTA,LTB)        /*FORM 3/16 S IN T*/
```

```
LTA:  ILR(Ø) FFØ JMP(LTC)          /*GET L IN AC*/
LTB:  ILR(1) FFØ
LTC:  ADR(T) FFØ JMP(OUT)          /*FORM L+3/16 S in T AND GO OUTPUT*/
 GT:  SRA(AC) FFØ                  /*S/L>1/2 : DIVIDE S BY 2*/
      SDR(T)  FF1                  /*STORE S/2 IN T*/
      SRA(AC) FFØ
      SRA(AC) FFØ                  /*FORM S/8 IN AC*/
      ADR(T)  FFØ                  /*FORM 5/8 S IN T*/
      SRA(AC) FFØ                  /*FORM S/16 IN AC*/
      ADR(T)  FFØ JCF(GTA,GTB)     /*FORM 11/16 S IN T*/
GTA:  ILR(Ø) FFØ  JMP(GTC)         /*GET L IN AC*/
GTB:  ILR(1) FFØ
GTC:  SRA(AC) FFØ                  /*DIVIDE L BY 2*/
      ADR(T) FFØ                   /*FORM L/2 + 11/16 S IN T*/

OUT:  LMI(T) FFØ OST JPX(OUT,START) /*OUTPUT RMS VALUE THRU MAR, GENERATE
                                     OUTPUT STROBE, & WAIT FOR DATA READY
                                     TO BEGIN A NEW CALCULATION*/
```

There are 42 microinstructions in the above program, but the longest path through it has 30 instructions, broken down as follows:

| | No. of Microinstructions |
|---|---|
| Fetch and take absolute values of I and Q | 9 |
| Determine larger magnitude | 3 |
| Compare L/2 with S | 7 |
| Form L/2 + 11/16 S | 10 |
| Output | 1 |
| TOTAL | 30 |

Assuming operation at the minimum rated cycle time of 125 ns for a pipelined machine, the worse case time for this algorithm is

$$30 \times 0.125\ \mu s = 3.75\ \mu s$$

Although this is considerably faster than the same algorithm programmed on a general purpose computer, it is much slower than the 5 MHz sampling rate. If this processor were to be used in the EAR system, first-in, first-out data buffers (FIFO) would be required on the I and Q input busses. In this case, the maximum number of range bins for which an RMS value could be computed would be

$$200\ \mu s/3.75\ \mu s = 53$$

for a sliding window MTI, or

$$600\ \mu s/3.75\ \mu s = 160$$

for a fixed window three pulse canceller.

Assuming a word length of 24 bits would be sufficient for this application, this microcomputer would require the following LSI packages:

| Description | Quantity |
|---|---|
| Intel 3001 MCU | 1 |
| Intel 3002 CPE 2 bit slice | 12 |
| Intel 3002 Look-Ahead Carry generator | 3 |
| 8 bit latch (for pipeline register) | 3 |
| 512 x 8 PROM or ROM (e.g. Intel 3604, 3304A, etc) for microprogram memory | 3 |
| 32 x 8 FIFO for input data buffering (160 I,Q pairs) | 15 |
| TOTAL | 37 |

Note that 512 words of microprogram memory are specified, even though only 42 words are used. This was done because of the previously mentioned memory assignment problem, which is eased when there are many more memory locations than microinstructions. Even if the number of words were reduced

to 256 or 128, it would still probably require at least 3 chips to obtain the 23 bit word length. The amount of FIFO specified assumes the worst case of a fixed window three pulse canceller and 160 range bins processed. Note that some random logic is required in addition to the above listed devices for clocking, K-bus decoding, etc.

In summary, the Intel 3000 Series is only marginally satisfactory in this application because of the very high data rate. Even though a 3000 Series-based machine can have a very short cycle time, the rather primitive microinstruction set and inherent sequential nature of the processing results in a computation cycle much longer than the sampling interval. In addition, microprogramming of the 3000 is a complicated and lengthy process.

### 3.2. A Programmable Logic Array Approach to the RMS Module

The primary drawback of the Intel 3000 implementation of the RMS module of the MICOM EAR Signal Processor was that it required an execution time of 3.75μs. This device was used in a pipelined configuration and its speed is near the top end of the state-of-the-art. The required execution time for the present EAR RMS timing is 200ns.

Preliminary results of the investigation into the use of PLA's indicate that the desirable properties of a microprocessor approach may be retained, that is,

a) programmable design,

b) flexibility and expandability, and

c) low parts count.

In addition the execution time looks like it will be much nearer 200ns.

The cost and nature of fabrication of PLA's make them less changeable than microprocessors. This does seem to be improving since Intersil now has available a programmable (rather than maskable) PLA. The cost of this chip is well under $100 in unit quantities.

A PLA Processing Module: Programmable Logic Arrays (PLA) have been used for some time to realize large blocks of combinational logic (see Fig. 3.4). This will not be the intent here. A register will be added to provide state variables which may be fed back as inputs. There are currently available chips which do this internally (TI, Collins); however, their delay times may be inadequate for the purposes defined here. Some PLA's currently available are given in Table 3.2

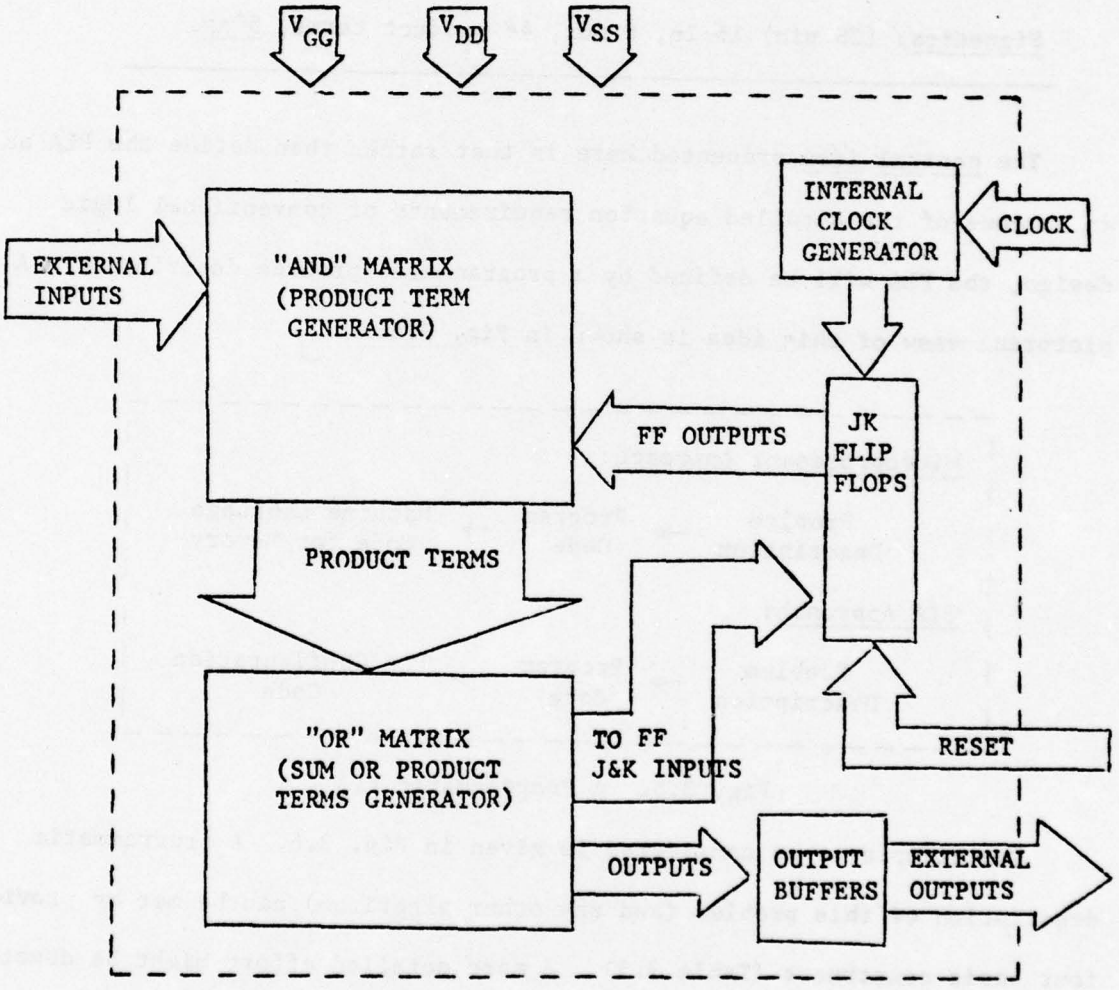



Fig. 3.4. Programmable Logic Array Block Diagram

Table 3.2. Available PLA's

TI: TMS 2000 (40 Pin)

17 In; 18 Out; 60 pr; 8 FF's (Internal)

TMS 2200 (28 Pin)

13 In; 10 Out; 72 pr; 10 FF's (Internal)

Collins: 16 In; 20 Out; 6 FF (Internal)

Rockwell: (48 Pin)

(I+O) $\leq$ 46; 128 product terms, 35ns

Signetics: (28 pin) 16 In, 8 Out, 48 product terms, 50ns.

---

The central idea presented here is that rather than define the PLA as an outcome of the detailed equation requirements of conventional logic design, the PLA will be defined by a programmatic problem description. A pictorial view of this idea is shown in Fig. 3.5.

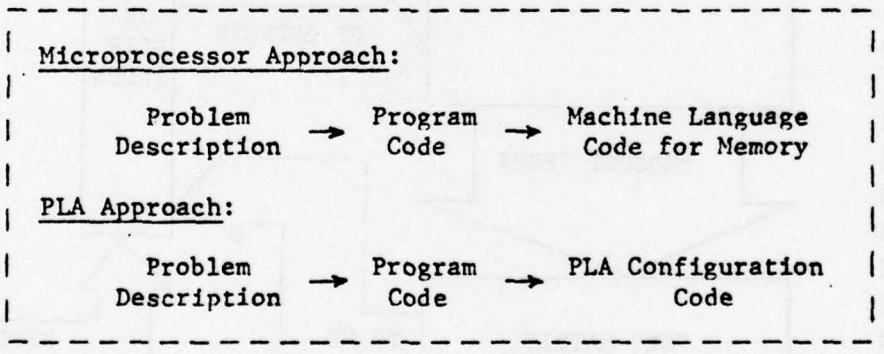



Fig. 3.5. A Programmatic PLA

The example to be considered is given in Fig. 3.6. A programmatic description of this problem (and any other algorithm) can be met by providing four basic constructs (Table 3.3). A more detailed effort might be devoted to the development of a programming language oriented to signal processing. The specifics of a language are not particularly germain to the ideas presented here.

Example: 'RMS'

IF $S/L \leq 0.5$ THEN $R = L + \frac{3}{16} S$;

ELSE $R = \frac{3L}{4} + \frac{11S}{16}$ ;

where $L = Max [I, Q]$

$S = Min [I, Q]$

Fig. 3.6. An Example Problem

Table 3.3. Basic Program Statements

DO UNTIL;

DO

IF THEN ELSE;

GO TO; (not essential)

What is central is that a compiler which takes a language description and produces suitable PLA descriptions is essential. It is the development of this compiler which is discussed and proposed here. The nature of the required output is shown in Table 2.3. It should be noted that the FF values given in Table 3.4 may be register (for example an accumulator) values as well as state variables.

Table 3.4. PLA Design Data

| External In | Present FF Values | External Outs | Next FF Values |
|---|---|---|---|
| | | | |

There are many ways to organize even this simple problem. The nature of problem parallelism and the use of pipelining are extremely important to the ultimate solution of a problem and the resulting execution time. These ideas will not be pursued here.

For the purposes of this example a single accumulator synchronous solution will be implemented. The basic algorithm is shown in Fig. 3.6.

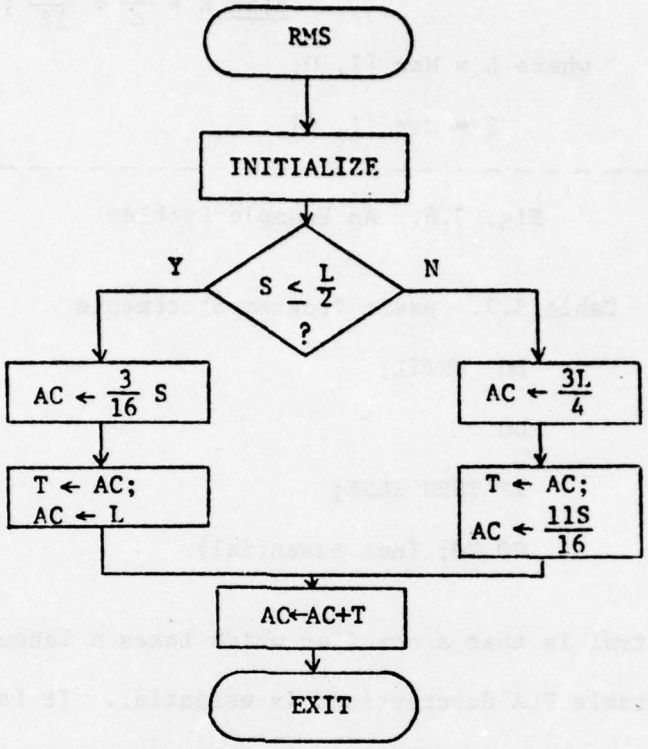



Fig. 3.6. A Single Accumulator Algorithm for RMS

A programmatic description is given in Table 3.5.

Table 3.5. Program for RMS Example

```
DO/*RMS*/

        RMS:  DO/*INITIALIZE*/;
              IF (S ≤ L/2) THEN CONTINUE, ELSE GO TO RMS1;
              DO AC ← 3S/16;
              DO T ← AC, AC ← L;
              GO TO RMS2;

       RMS1:  DO AC ← 3L/4 ;

              DO T ← AC, AC ← 11S/16 ;

       RMS2:  DO AC ← AC + T;
              END;
END;
```

Now to see what the PLA design information looks like. The following data types will be assumed to simplify the problem size. No generality of the concepts is lost in this assumption (Table 3.6). In addition to the data in Table 3.6 there will be state variables. All FF's will be synchronously clocked.

Table 3.6. Data Designations

| | |
|---|---|
| L | 2 bits unsigned (EXT IN) |
| S | 2 bits unsigned (EXT IN) |
| T | Temporary External Register<br>4 bits unsigned (FF value) |
| AC | External Accumulator Register<br>4 bits unsigned (FF value) |

It should be noted that in generating the PLA design data required for this problem (Table 3.7) considerable insight is required. This will ultimately be the task of a compiler. An algorithmic approach to this table generation is not known at this time; however, a considerable amount has been learned in the process of doing the work to date. Since the table requires $2^{15}$ entries to define exhaustively it is given in shorthand notation. From the information given in Table 3.7 it can be seen that a 15 in and 11 out PLA and 11 FF are required. For practical dimensions on this problem's data no single PLA could meet the needs.

PLA's are easily interconnected to realize large problems, however some practical algorithm for describing this expansion is required. Note that the state variables do not increase as the data width increases.

The information in Table 3.7 can be expanded to give the total design parameters for a PLA. This process, while not yet strictly algorithmic, does demonstrate the feasibility of the approach.

Now, how about execution time? The longest path requires four clock cycles to increment the PLA through the algorithm. With a PLA delay of 35ns

Table 3.7. PLA Design Data

| | EXT IN | | FF IN | | | FF OUT | | |
|---|---|---|---|---|---|---|---|---|
| | I1 2 | Q1 2 | T1 2 3 4 | AC1 2 3 4 | SV1 2 3 | T1 2 3 4 | AC1 2 3 4 | SV1 2 3 |
| Y | 0 0 | 0 0 | X X X X | X X X X | 0 0 0 | X X X X | X X X X | 0 0 1 |
| Y | 0 0 | 0 1 | | | | | | 0 0 1 |
| Y | 0 0 | 1 0 | | | | | | 0 0 1 |
| Y | 0 0 | 1 1 | | | | | | 0 0 1 |
| Y | 0 1 | 0 0 | | | | | | 0 0 1 |
| N | 0 1 | 0 1 | | | | | | 0 1 1 |
| Y | 0 1 | 1 0 | | | | | | 0 0 1 |
| Y | 0 1 | 1 1 | | | | | | 0 0 1 |
| Y | 1 0 | 0 0 | | | | | | 0 0 1 |
| Y | 1 0 | 0 1 | | | | | | 0 0 1 |
| N | 1 0 | 1 0 | | | | | | 0 1 1 |
| N | 1 0 | 1 1 | | | | | | 0 1 1 |
| Y | 1 1 | 0 0 | | | | | | 0 0 1 |
| Y | 1 1 | 0 1 | | | | | | 0 0 1 |
| N | 1 1 | 1 0 | | | | | | 0 1 1 |
| N | 1 1 | 1 1 | X X X X | X X X X | 0 0 0 | X X X X | X X X X | 0 1 1 |
| | I | Q | X X X X | X X X X | 0 0 1 | X X X X | $\frac{3}{16}$(min(I,Q)) | 0 1 0 |
| | I | Q | X X X X | $\frac{3}{16}$(min) | 0 1 0 | $\frac{3}{16}$(min) | max(I,Q) | 1 0 1 |
| | I | Q | X X X X | X X X X | 0 1 1 | X X X X | $\frac{3}{4}$(max) | 1 0 0 |
| | I | Q | X X X X | $\frac{3}{4}$(max) | 1 0 0 | $\frac{3}{4}$ (max) | $\frac{11}{16}$(min) | 1 0 1 |
| | I | Q | T prev. | AC prev. | 1 0 1 | T in | T in+AC in | 0 0 0 |

and a FF delay about the same the total delay is 280ns. This time does not increase with an increase in the data width. Other solution approaches are possible and some may have better execution times.

With respect to the original design goals, the package count has not been discussed. This of course greatly depends upon the particular device selected. The critical concerns are input/output pins and number of product terms possible. In order to decompose the problem into realistic modules the organization of the algorithm must be reconsidered. This step is critical from the point of view of pin constraints, product term constraints and propagation delay.

In order to estimate the pin and ultimately package requirement the interface to the module and to the PLA must be defined. A module block diagram with this information is given in Fig. 3.7.

Using the data width requirements of Fig. 2.5, the PLA pin requirements are

| | |
|---|---|
| PLA In pins | 19+19+19+24+3 = 84 |
| PLA Out pins | 19+24+3 = 46 |
| | TOTAL: 130 pins |

In order to make an estimate the problem must be considered in more detail since the process of actual decomposition may well create the need for additional internal signals and levels of logic. At this point in time there are no systematic approaches to decomposing this large of a problem. The following discussion outlines one possible implementation and identifies some of the problems.

The computation of the magnitude over 19 bits is difficult to segment since the entire value must be examined at one time. The easy out is to use one PLA per input vector (one for I and one for Q). This requires two packages of the largest variety (38 pins total).

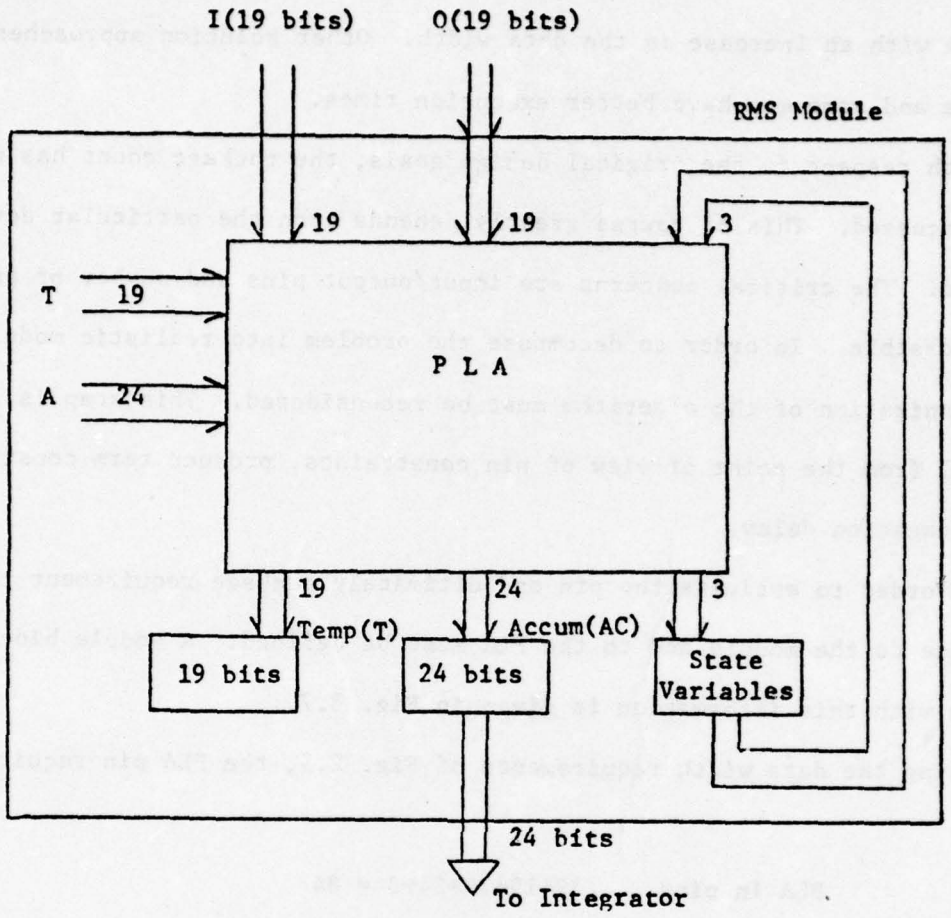



Fig. 3.7. RMS Module Block Diagram

Now the computation can be reformatted to be a selection of one of four possible. This avoids the requirement for steering and is shown in Table 3.8. Viewed as shift and add computation each case can be rewritten as in Table 3.9. I and Q are magnitude values throughout. The use of a PLA motivates the computation of these values as combinational and not sequential processes. This motivates the decomposition shown in Fig. 3.8. The shifting is accomplished by feeding right the maximum number of shift places (4 here). In this format all of the required fractional values are available at the module. The computation now reduces to an add of the proper selected values. The size of the add segment may be computed by examining the detailed picture of Fig. 3.9.

Table 3.8. Possible Computations for RMS

| IF $\frac{I}{Q}$ | | Case |
|---|---|---|
| $\leq .5 \rightarrow$ | $\lvert I \rvert = S;\ \lvert Q \rvert = L$ | |
| | $R = \lvert Q \rvert + \frac{3}{16} \lvert I \rvert$ | 0 |
| $> .5 \rightarrow$ | $\lvert I \rvert = S;\ \lvert Q \rvert = L$ | |
| but $< 1$ | $R = \frac{3}{4} \lvert Q \rvert + \frac{11}{16} \lvert I \rvert$ | 1 |
| $\geq 1.0 \rightarrow$ | $\lvert I \rvert = L;\ \lvert Q \rvert = S$ | |
| but $< 2.0$ | $R = \frac{3}{4} \lvert I \rvert + \frac{11}{16} \lvert Q \rvert$ | 2 |
| $\geq 2.0 \rightarrow$ | $\lvert I \rvert = L;\ \lvert Q \rvert = S$ | |
| | $R = \lvert I \rvert + \frac{3}{16} \lvert Q \rvert$ | 3 |

Table 3.9. RMS Computations

Case 0 $R = Q + 1/8\ I + 1/16\ I$

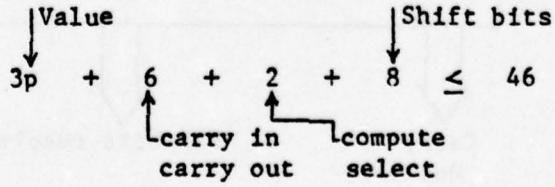
Case 1 $R = 1/2\ Q + 1/4\ Q + 1/2\ I + 1/8\ I + 1/16\ I$

Case 2 $R = 1/2\ I + 1/4\ I + 1/2\ Q + 1/8\ Q + 1/16\ Q$

Case 3 $R = I + 1/8\ Q + 1/16\ Q$

where each fraction is a shift right of 1, 2, 3 or 4.

First, the number of carry bits required must be computed. The worst case is represented by the computation of cases 1 or 2. This is an add of 5 operands which may generate a maximum carry of 4 or, that is, 3 bits. Using the 46 pin module would require that the following equation hold.




or a maximum module width of p=10 bits. Since the maximum result possible will have 21 bits two modules will suffice as far as the pin requirement is concerned.

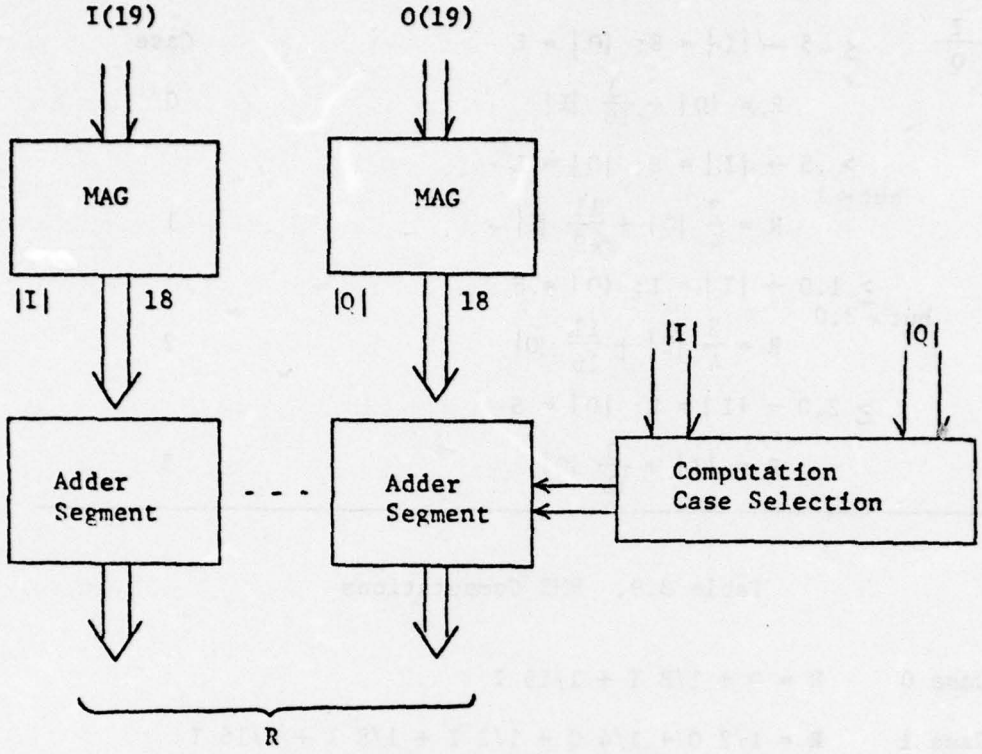



Fig. 3.8. RMS Decomposition

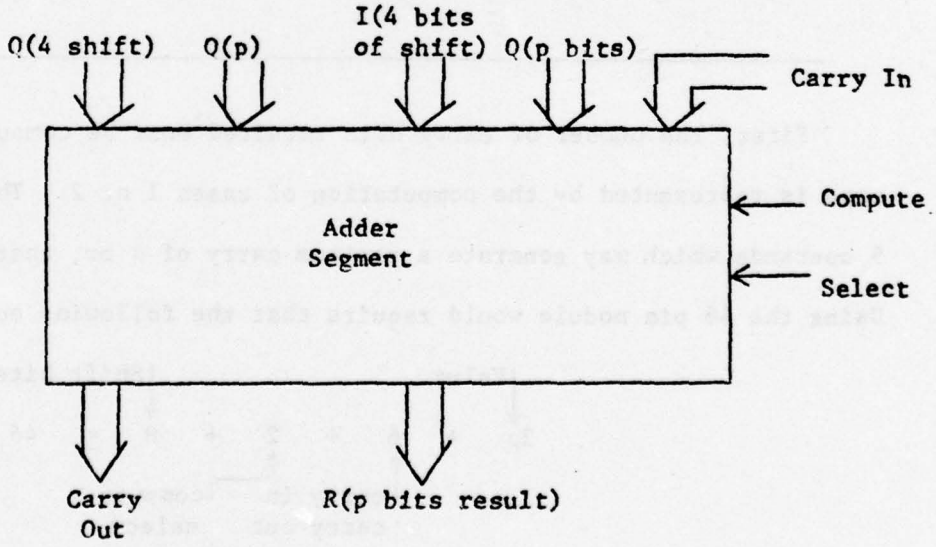



Fig. 3.9. Adder Module Detail

This does not answer the question concerning the number of product terms required. The ultimate answer to this question can best be approached by a computer oriented algorithm using well established minimization procedures. These programs have not been utilized as of this writing. It is possible to add additional chips in parallel to expand the product terms available. A practical limit to this expansion obviously is reached quite early. In the limit a ROM is required if all product terms are min terms. This question cannot be pursued within the time frame of this study. A configuration based on pin restrictions is given in Fig. 3.10. This represents a lower bound on package count which may be expanded due to product term requirements.

It should be noted that this solution is not sequential and does not follow the earlier programmatic description. The worse case delay is through four modules for a total of approximately 140ns using the Rockwell chip.

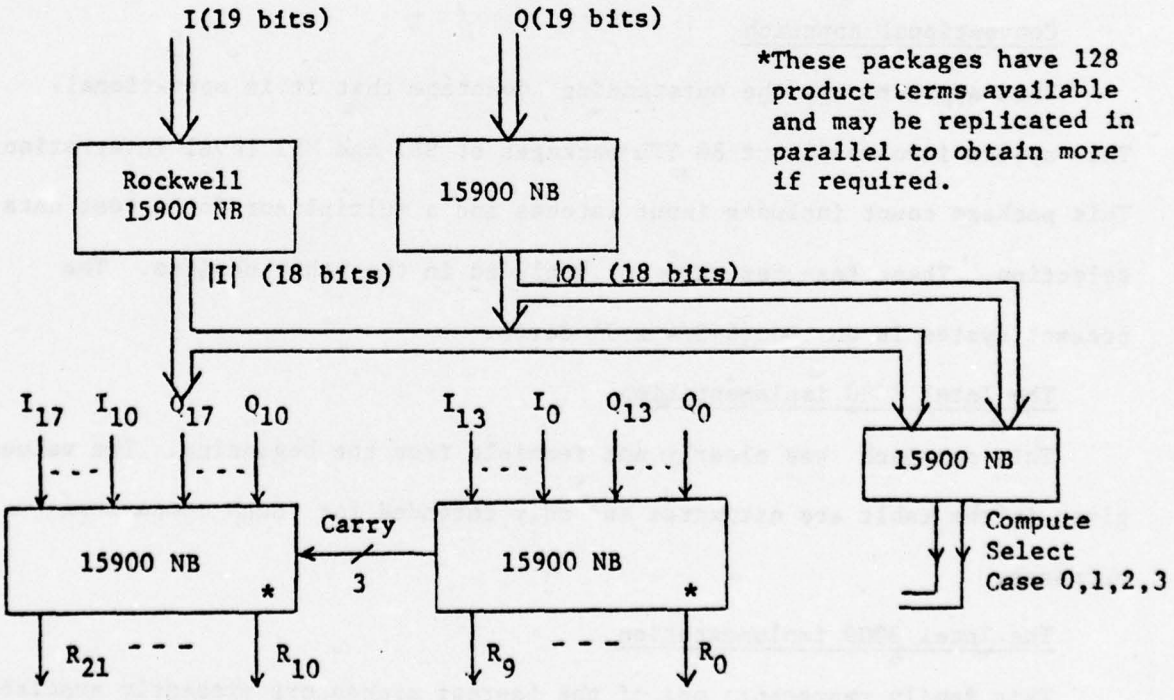



Fig. 3.10. Detailed RMS Pin Constraint Solution

## 3.3. Comparison of Approaches to the RMS Module

In making a comparison of several approaches to a design it is quite difficult to keep the comparison relatively consistent. The approaches to be compared are

(a) The conventional (existing) logic implementation,

(b) The Intel 8080 8 bit processor,

(c) The Intel 3000 bipolar slice processor,

(d) A PLA implementation.

The important aspects of each design which will be compared are

(a) parts count,

(b) delay time,

(c) design approach, and

(d) flexibility.

A short description of each approach follows and the parameters are summarized in Table 3.10.

Conventional approach

This approach has the outstanding advantage that it is operational. This system involves about 80 TTL packages of SSI and MSI level integration. This package count includes input latches and a multiplexor for a test data selection. These features were not included in the other designs. The present system is on two (6-3/4 x 9) cards.

The Intel 8080 implementation

This approach was clearly not feasible from the beginning. The values given in the table are estimates and only intended for rough comparison purposes.

The Intel 3000 implementation

This family represents one of the fastest processors presently available. The configuration which was considered utilized pipelining and omitted the

use of main memory. The figures given are very representative of the best speed which can be achieved with present technology.

A PLA implementation

The PLA offers considerable promise in terms of providing large scale integration with speeds competative with conventional logic. The design procedures are not well understood at this time. In an experimental environment field programmable devices are more practical than masked devices. Unfortunately the size of field programmable units is much smaller, thus raising the package count. The values given for the PLA in Table 3.10 must be taken as fairly optimistic bounds. This approach needs considerable additional study.

Table 3.10. Summary of Results for RMS Module Implementation

| Approach | Packages | Execution Time | Flexibility | Design Methodology |
|---|---|---|---|---|
| Conventional Logic | 60 MSI/SSI w/o input packs | 170 ns | poor | familiar |
| 8080 8 bit processor | 25-30 MSI/LSI packs | approx. 1 ms for 18 bit operands | very good | Systematic - mostly programming |
| 3000 bipolar slice | approx. 40 with buffering | 3-4 $\mu$s | fair (ROM prog size limit) | Systematic - tougher programming |
| Programmable logic array (PLA) | 6-10 lower bound | about 5 delays @ 35-50 ns 200-250 ns | fair (prod. term and pin limits) | Evolving - |

# 4. THE ANGLE TRACK COMPUTER (ATRAC)
## A Case Study

With the experience of the last section it seemed desirable to pursue a similar examination on a more complex module. The ATRAC was selected for a number of reasons:

(a) the design must be updated for the new system,

(b) the module is well defined and at about the right degree of complexity, and

(c) adequate documentation exists.

### 4.1. The Present System

The intent of this section is to summarize the essential aspects of the ATRAC requirements as they relate to the case study design at hand. The ATRAC is a special purpose computer designed to compute the antenna position error given by

$$\epsilon = \frac{1}{16} \sum_{K} \left( \frac{\Sigma_{IK} \Delta_{IK} + \Sigma_{QK} \Delta_{QK}}{\Sigma_{IK}^2 + \Sigma_{QK}^2} \right)$$

The present timing requirements are as follows:

PRF = pulse repetition frequency $5=10^3$ Hz (200 μs period)

DWELL TIME = 50 PRF periods (10 ms)

INPUT DATA RATE = From the MTI filter. Presently a three pulse canceller producing a residue at a 600 μs interval for a total of 16 residues

DATA SIZE = Currently 11 bits for both I, Q inputs and 15 bits out

ATRAC CYCLE TIME = 400 ns

### 4.2. Projected Changes

The next generation implementation of the ATRAC is likely to incorporate changes in both the data widths and in timing. The following design example will consider these potential changes.

Input data rate = approaching one per PRF or every 200 μs.

Input data width = 18 bits plus sign

Output data width = 23 bits plus sign

One of the objectives of the proposed design approaches is to provide the flexibility to handle various data rates and widths.

## 4.3. ATRAC Intel 8080 Implementation

It seems fairly clear that since the 8080 RMS module required about 1 ms of computation time it will not be possible to meet the time demands with a single 8080 module. It may, however, be possible to segment the ATRAC computation and meet the basic 200 μs time slot requirement.

The multi segment pipeline of Fig. 4.1 is about the smallest practical problem segmentation. This pipeline has four sections requiring a total of eight processing elements. The execution demand for each segment is 200 μs. This approach would utilize a total of eight 8080 processors and associated hardware.

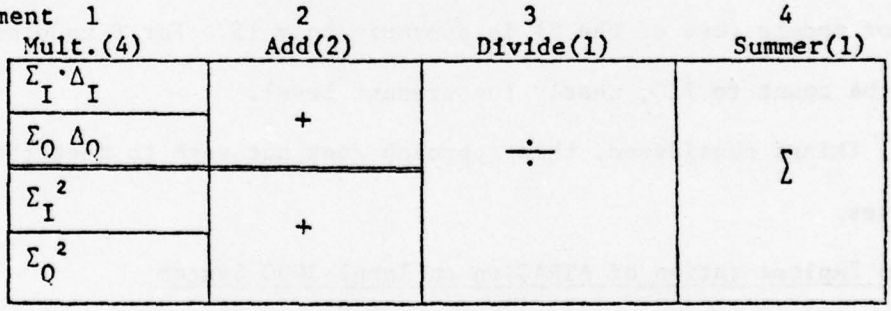



Fig. 4.1. Problem Pipeline Organization

The throughput which can be obtained using this configuration is bounded by the slowest segment. Processing time estimates are given in Table 4.1.

Table 4.1. 8080 Computation Time Estimates

| Segment | I/O | Compute | Total |
|---|---|---|---|
| 1 (mult) | 120 μs | 1050 μs | 1.2 ms |
| 2 (adder) | 120 μs | 120 μs | .28 ms |
| 3 (divide) | 120 μs | 1250 μs | 1.4 ms |
| 4 (summer) | 120 μs | 120 μs | .28 ms |

Each segment has an estimated In/Out overhead time of 127 μs plus its own internal computation requirement. Without additional buffering the divide segment restricts the processing to 7 residues per dwell time slot.

There are a number of additional possibilities to speed up the process, however they seem to all fall short of the required improvement required.

a) Data buffering in a FIFO -- there is no significant time when catch up could occur.

b) Direct memory access -- this would improve the In/Out time requirement, however this time is quite small compared to computation.

c) Faster external hardware -- the division and multiply would need to be implemented in external hardware. A single processor module might then be used for addition and summing. This would then seem to be so near to the conventional logic approach that little would be gained by using a microprocessor at all.

Another discouraging point is that the chip count for a minimal microprocessor module (one of the 8) is approximately 15. For 8 modules this brings the count to 120, nearly the present level.

All things considered, this approach does not seem to meet the basic objectives.

## 4.4. An Implementation of ATRAC on an Intel 3000 System

This section describes in detail an implementation of the present ATRAC algorithm on the Intel 3000 signal processing microcomputer proposed earlier (Fig. 3.2) with reference to the RMS algorithm. This system is shown to be considerably faster than the existing ATRAC hardware. Projected expansions in word length are discussed; these leave computation times still well within the requirements. Estimated package counts are given.

A basic decision was made to make the CPE word length at least twice the input word length, e.g., 22 bits for the present ATRAC. This allows

each of the four products to be retained at full precision in a single CPE register. Since there is no truncation to 16 bits as in the existing ATRAC hardware, there is no longer a need to normalize the numerator and denominator of each residue and scale the quotient. This fact is partly responsible for the increased speed of the 3000 implementation.

The basic sequence of operations programmed on the 3000 system is as follows:

1) Wait for DTI pulse to start the algorithm.
2) Clear $\varepsilon$ accumulator and set residue counter.
3) Wait for $\Sigma$ strobe.
4) Input $\Sigma_I$, $\Sigma_Q$ (assumed in 2's complement representation)
5) Input $\Delta_I$, $\Delta_Q$
6) Form the four products $\Sigma_I \Delta_I$, $\Sigma_Q \Delta_Q$, $\Sigma_I^2$ and $\Sigma_Q^2$. Each is retained to full precision: 20 bits magnitude plus sign for 10 bit magnitude plus sign input data.
7) Form numerator and denominator by summing first and second pairs of products above.
8) Shift numerator right by 4 bits (divide by 16). This assures that for the normal operating range in which $|\varepsilon|<2$, the magnitude of the divident will be less than twice the divisor, as required for proper operation of the division algorithm.
9) Divide denominator into numerator. Form quotient to 18 bits magnitude plus sign. This is 4 bits greater than the output word length (14 bits magnitude plus sign) so that, when the 16 residues are summed, truncation errors will not accumulate beyond the LSB of the 15 bit output word.
10) Add the quotient to the $\varepsilon$ accumulator.

11) Increment the residue counter. If this was the sixteenth residue, output ε, generate an output strobe, and go to step 1. Otherwise, go to step 3.

The multiplication microcode used is somewhat simpler than the example given on page 9 of Intel Apllication Note AP-13 because the multiplier and multiplicand magnitudes each occupy less than half the CPE word length. Hence the product requires only a single register, rather than two registers as in Intel's example. The algorithm is the common shift-and-add method. The code is listed below:

```
     /*Enter with multiplier magnitude in lower half of T*/
     /*and multiplicand magnitude in lower half of AC*/
     /*R8 is used for a loop counter*/
     /*The product is accumulated in R7*/
     CSR(R8) FFØ                        /*Set R8 to all 1's*/
     TZR(R8) KNMS                       /*Set R8 to negative of multiplier*/
                                        /*word length entered via K-bus*/
     CLR(R7)                            /*Clear R7 for product*/
     SRA(T)                             /*Fetch multiplier LSB*/
     /*Increment loop count, save its overflow in Z-flag, and test multiplier LSB*/
MLP: INR(R8) FF1 STZ JFL(MBZ,MB1)
     /*If LSB=1, add multiplicand to partial product*/
MB1: ADR(R7) FFØ
MBZ: ALR(AC) FFØ                        /*Shift multiplicand left 1 bit*/
     SRA(T) JZF(MLP,MEX)                /*Get next multiplier LSB and
                                        /*loop if loop count not full*/
MEX: . . . . .
```

Note that the multiply loop consists of 3 or 4 instructions per bit of the multiplier.

The division algorithm is quite similar to that used in the present ATRAC hardware and is listed below:

```
      /*Enter with dividend magnitude in R9*/
      /*and negative of divisor magnitude in R8*/
      /*For proper operation, the dividend magnitude*/
      /*must be less than twice the divisor magnitude*/
      /*The quotient is generated in R7*/
      /*R6 is used as a loop counter*/
      CSR(R6) FFØ                        /*Set R6 to all 1's*/
      TZR(R6) KNDLC                      /*Set R6 to negative of divide*/
                                         /*loop count, i.e., no. of bits in quotient*/
      CLR(R7)                            /*Clear quotient register*/
DLUP: ILR(R8) FFØ                        /*Get negative divisor in AC*/
      /*Get sign of dividend-divisor (1 if positive) and save
      /*in C-flag, but inhibit loading of difference:*/
      ADR(R9) FFØ STC INH
      /*Increment loop counter, save its overflow in Z-flag*/
      /*and test sign of dividend-divisor:*/
      INR(R6) FF1 STZ JFL(DNEG,DPOS)
      /*If dividend-divisor is non-negative, form the difference*/
      /*again and save in R9 and AC:*/
DPOS: ALR(R9) FFØ JMP(DLS)
      /*If the difference is negative, get dividend in AC:*/
DNEG: ILR(R9) FFØ
 DLS: ALR(R9) FFØ                        /*Shift new dividend left and store in R9*/
      ILR(R7) FFØ                        /*Get partial quotient in AC*/
      /*Shift partial quotient left 1 bit and add new quotient*/
      /*bit from C-flag to LSB.  Test loop count:*/
```

```
    ALR(R7) FFC JZF(DLUP,DEX)
DEX: . . . .
```

Note that the divide loop consists of 7 instructions per quotient bit.

Before the microcode for the ATRAC algorithm is given the details of data input and output to and from the microcomputer are described. Output is straightforward: There is only one variable, $\varepsilon$, which is output after the accumulation of 16 residues, i.e., once per dwell time. The CPE's MAR (main memory address) port was chosen for this function since it can latch $\varepsilon$ and hold it during the next computation cycle. The four least significant bits of this output should not be used because of accumulated truncation error. (Recall that each quotient is generated to four more bits than the desired word length of $\varepsilon$.) One bit of the microinstruction field is used as an output strobe as in the RMS algorithm implementation.

Data input on the other hand, is somewhat more complicated because the $\Sigma$ and $\Delta$ values appear sequentially on the same bus 400 ns apart, separated by one unused sample. That is, each sample is valid on the bus for less than 200 ns. Since the microinstruction cycle is about 125 ns and since the sample and microprocessor clocks will probably be asynchronous, it is clear that input data buffering external to the CPE is required. The scheme described below does this with a minimum amount of extra hardware.

Since the CPE word length was chosen to be at least twice the input word length, the I and Q busses may be brought into a single CPE input port, say the I port. This saves one input microinstruction during the time-constrained input routine. If $\Sigma_I$ and $\Sigma_Q$ are latched, it is then possible to input them into the CPE prior to the arrival of $\Delta_I$ and $\Delta_Q$ 400 ns later. Thus only 22 bits worth of latches are required, as shown in Fig. 4.2. Note that strobe generating logic similar to that in the existing ATRAC hardware is required.

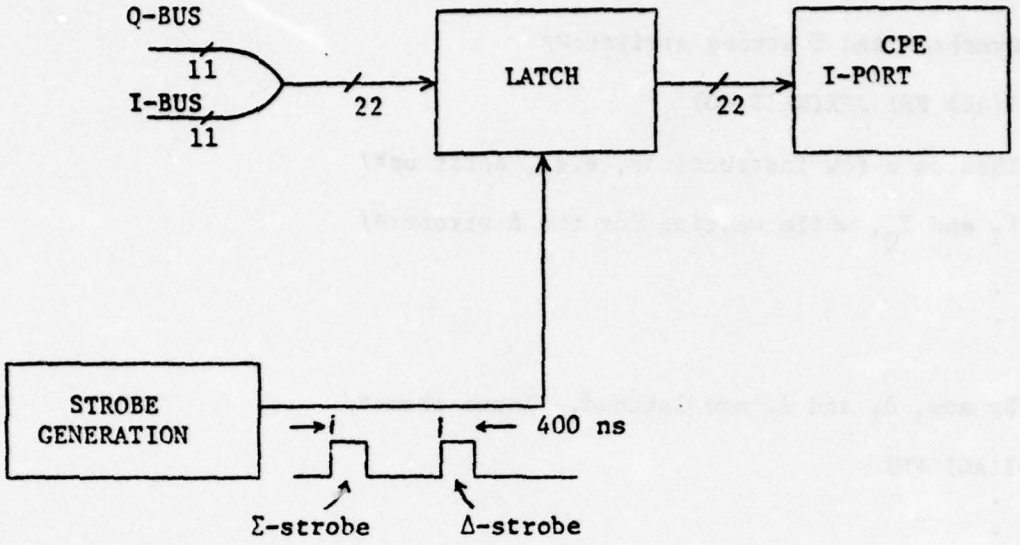



Fig. 4.2. Input Data Latching

The first strobe of the pair (Σ) must cause a conditional jump in the MCU to begin the computation cycle. One of the primary instruction inputs X4-X7 may be used for this purpose. However, this input must be synchronized to the MCU clock. A suggested scheme for doing this is shown in Fig. 4.3.

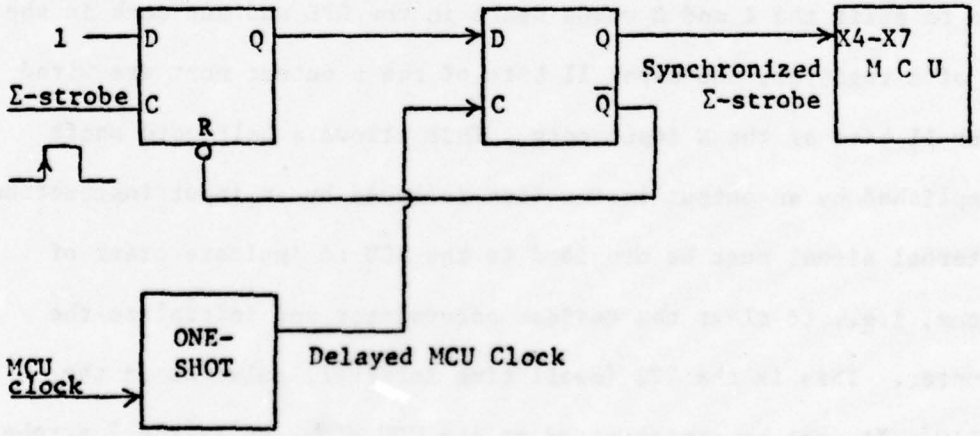



Fig. 4.3. Input Strobe Synchronization

The following microcode should be used with the configuration described:

```
     /*Keep inputting from the I-port until the*/
     /*synchronized Σ strobe arrives:*/
WAIT: LDI(AC) FF1 JPX(WAIT,GO)
     /*Then do a few instructions, e.g., split up*/
     /*Σ_I and Σ_Q, while waiting for the Δ strobe:*/
  GO:    .
         .
         .
     /*By now, Δ_I and Δ_Q are latched.  Input them:*/
     LDI(AC) FF1
         .
         .
         .
```

The timing diagram of Fig. 4.4 shows that under worst case conditions, the latched Σ data are input to the CPE approximately 100 ns prior to latching of the Δ data. (Note that the pipelined processor configuration delays the CPE instruction (data input) one microcycle with respect to the MCU instruction (conditional jump on strobe).)

In order to eliminate the 11 single bit shift instructions which would be required to split the I and Q words apart in the CPE and get each in the lower half of a register, the upper 11 bits of the D output port are wired to the lower 11 bits of the M input port. This allows a half word shift to be accomplished by an output instruction followed by an input instruction.

An external signal must be provided to the MCU to indicate start of the algorithm, i.e., to clear the residue accumulator and initialize the residue counter. This is the DTI (dwell time interval) pulse as in the present ATRAC. It must be synchronized to the MCU clock as is the Σ strobe (see Fig. 4.3) and applied to one of the MCU primary instruction inputs X4-X7 to trigger a conditional jump.

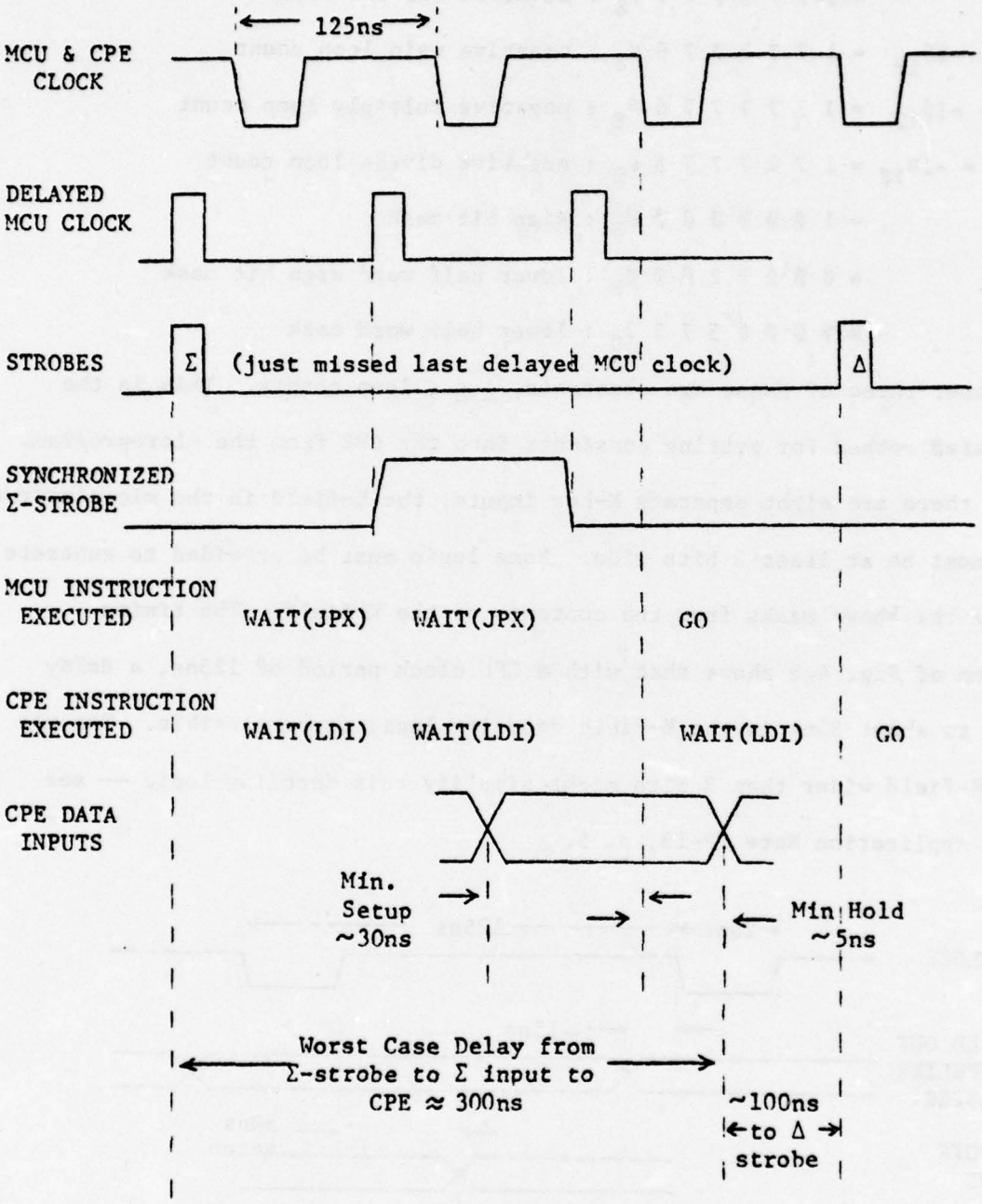



Fig. 4.4. Input Timing

Finally, the ATRAC microcode required the following CPE K-bus inputs:

KØ = Ø Ø Ø Ø Ø Ø Ø $Ø_8$ : standard all-zero mask

K1 = 1 7 7 7 7 7 7 $7_8$ : standard all-one mask

KNLC = $-16_{1Ø}$ = 1 7 7 7 7 7 6 $Ø_8$ : negative main loop count

KNMC = $-1Ø_{1Ø}$ = 1 7 7 7 7 7 6 $6_8$ : negative multiply loop count

KNDLC = $-18_{1Ø}$ = 1 7 7 7 7 7 5 $6_8$ : negative divide loop count

KSB = 1 Ø Ø Ø Ø Ø Ø $Ø_8$ : sign bit mask

KLHSB = Ø Ø Ø Ø 2 Ø Ø $Ø_8$ : lower half word sign bit mask

KLHW = Ø Ø Ø Ø 3 7 7 $7_8$ : lower half word mask

Note that three of these are constants, i.e., loop counts. This is the suggested method for getting constants into the CPE from the microprogram. Since there are eight separate K-bus inputs, the K-field in the microinstruction must be at least 3 bits wide. Some logic must be provided to generate one of the above masks from the contents of the K-field. The timing diagram of Fig. 4.5 shows that with a CPE clock period of 125ns, a delay of up to about 35ns in the K-field decoding logic is permissible. The use of a K-field wider than 3 bits might simplify this decoding logic — see Intel Application Note AP-13, p. 5.

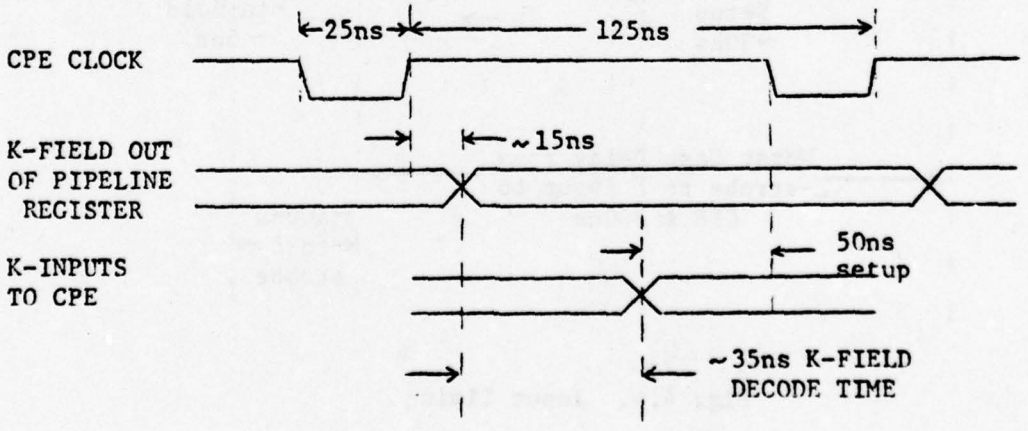



Fig. 4.5. K-Field Decode Timing

The complete microcode for the implementation of the present ATRAC algorithm on the Intel 3000 pipelined processor configured as described above is given below:

```
WT1: CLR(RØ) JPX(WT1,GO1)          /*Clear RØ: accumulator for*/
                                   /*the 16 residues.  Wait for DTI*/
                                   /*pulse to start algorithm*/
GO1: CSR(R1) FFØ                   /*Set R1 to all 1's*/
     TZR(R1) KNLC                  /*Set R1 to -16 main loop count*/
WT2: LDI(AC) FF1 JPX(WT2,GO2)      /*Keep inputting from I-bus until*/
                                   /*Σ strobe arrives*/
GO2: SDR(R2) FF1                   /*Σ_Q, Σ_I in AC.  Store Σ_I in lower*/
                                   /*half of R2*/
     ACM(AC) FFØ                   /*Get Σ_Q in lower AC*/
     SDR(R4) FF1                   /*Save Σ_Q in R4*/
     LDI(AC) FF1                   /*Get Δ_Q & Δ_I in AC*/
     SDR(R3) FF1                   /*Save Δ_I in lower R3*/
     ACM(AC) FFØ                   /*Get Δ_Q in lower AC*/
     SDR(T) FF1                    /*Save Δ_Q in T*/
     CSR(R9) FFØ                   /*Set R9 to -1(for sign storage)*/
     ILR(R4) FFØ                   /*Get Σ_Q in AC*/
     TZR(AC) FFØ KLHSB INH         /*Get sign bit of Σ_Q*/
     TZR(T) FFØ KLHSB INH JFL(SQP,SQN)*/
                                   /*Get sign bit of Δ_Q and test sign of Σ_Q*/
SQP: INR(R9) FF1 JFL(DQP,DQN)      /*Increment R9 if Σ_Q positive*/
                                   /* & test sign of Δ_Q*/
SQN: CIA(AC) FF1 JFL(DQP,DQN)      /*Complement Σ_Q  if negative & test sign of Δ_Q*/
DQP: INR(R9) FF1 JMP(SQMSK)        /*Increment R9 if Δ_Q pos*/
DQN: CIA(T) FF1                    /*Complement Δ_Q if neg*/
```

```
SOMSK: TZR(AC) KLHW                    /*Mask out garbage in upper half of AC*/
       SDR(R4) FF1                     /*Save magnitude of Σ_Q in R4*/
       /*Form Σ_Q Δ_Q product:*/
       CSR(R5) FFØ                     /*Set R5 to all 1's*/
       TZR(R5) KNMC                    /*Set R5 to negative multiply loop count(-1Ø)*/
       CLR(R7)                         /*Clear R7 for product*/
       SRA(T)                          /*Fetch multiplier (Δ_Q) low order bit*/
SDQML: INR(R5) FF1 STZ JFL(SDQZ,SDQ1) /*Increment loop count, save its */
                                       /*overflow in Z-flag, & test multiplier*/
                                       /*low order bit*/
 SDQ1: ADR(R7) FFØ                     /*If this bit set, add multiplicand (Σ_Q)*/
                                       /*to partial product*/
 SDQZ: ALR(AC) FFØ                     /*Shift multiplicand left 1 bit*/
       SRA(T) JZF(SDQML,ISGNS)         /*Get next low order bit of */
                                       /*multiplier & test loop count*/
ISGNS: CSR(R8) FFØ                     /*Set R8 to -1 for sign storage*/
       ILR(R3) FFØ                     /*Δ_I → AC*/
       SDR(T) FF1                      /*Δ_I → T*/
       ILR(R2) FFØ                     /*Σ_I → AC*/
       TZR(AC) FFØ KLHSB INH           /*Get sign bit of Σ_I*/
       TZR(T) FFØ KLHSB INH JFL(SIP,SIN)
                                       /*Get sign bit of Δ_I and test sign of Σ_I*/
  SIP: INR(R8)FF1 JFL(DIP,DIN)         /*Increment R8 if Σ_I pos*/
                                       /*and test sign of Δ_I*/
  SIN: CIA(AC) FF1 JFL(DIP,DIN)        /*Complement Σ_I if negative*/
                                       /*and test sign of Δ_I*/
  DIP: INR(R8) FF1 JMP(SIMSK)          /*Increment R8 if Δ_I pos*/
  DIN: CIA(T) FF1                      /*Complement Δ_I if negative*/
```

```
SIMSK: TZR(AC) KLHW                    /*Mask out garbage in upper AC*/
       SDR(R2) FF1                     /*Save magnitude of Σ_I in R2*/
       /*Form Σ_I Δ_I product*/
       CSR(R5) FFØ                     /*Set R5 to all 1's*/
       TZR(R5) KNMC                    /*Set R5 to negative mpy loop count*/
       CLR(R6)                         /*Clear R6 for product*/
       SRA(T)                          /*Fetch multiplier (Δ_I) low order bit*/
SDIML: INR(R5) FF1 STZ JFL(SDIZ,SDI1)  /*Increment loop count, save its*/
                                       /*overflow in Z-flag and test multiplier*/
                                       /*low order bit*/
 SDI1: ADR(R6) FFØ                     /*If bit set, add multiplicand (Σ_I*/
                                       /*to partial product*/
 SDIZ: ALR(AC) FFØ                     /*Shift multiplicand left 1 bit*/
       SRA(T) JZF(SDIML,DNOM)          /*Get next LSB of multiplier*/
                                       /* and test loop count*/
       /*Form Σ_Q^2 + Δ_I^2*/
 DNOM: CLR(R3)                         /*Clear R3 for denominator*/
       CSR(R5) FFØ                     /*Set R5 to all 1's*/
       TZR(R5) KNMC                    /*Set R5 to negative mpy loop count*/
       ILR(R4) FFØ                     /*Get magnitude of Σ_Q in AC*/
       SDR(T) FF1                      /*and in T*/
       SRA(T)                          /*Get multiplier low order bit*/
SSQML: INR(R5) FF1 STZ JFL(SSQZ,SSQ1)  /*Increment loop count,*/
                                       /*save its overflow in Z-flag, and*/
                                       /*test multiplier low order bit*/
 SSQ1: ADR(R3) FFØ                     /*If bit set, add multiplicand to*/
                                       /*partial product*/
 SSQZ: ALR(AC) FFØ                     /*Shift multiplicand left 1 bit*/
       SRA(T) JZF(SSQML,SSIPR)         /*Get next LSB of multiplier*/
                                       /*& test loop count*/
```

```
SSIPR: CSR(R5) FFØ                    /*Reset loop count for*/
       TZR(R5) KNMC                   /*Σ_I Σ_I product*/
       ILR(R2) FFØ                    /*Get magnitude of Σ_I in AC*/
       SDR(T) FF1                     /*and in T*/
       SRA(T)                         /*Get multiplier LSB*/
SSIML: INR(R5) FF1 STZ JFL(SSIZ,SSI1) /*Increment loop*/
                                      /*count, save its overflow in Z-flag*/
                                      /*& test multiplier low order bit*/
 SSI1: ADR(R3) FFØ                    /*If bit set, add multiplicand to*/
                                      /*Partial product*/
 SSIZ: ALR(AC) FFØ                    /*Shift multiplicand left 1 bit*/
       SRA(T) JZF(SSIML,NUSUM)        /*Get next LSB of multiplier &*/
                                      /*test loop count*/
       /*Form numerator:*/
NUSUM: ILR(R6) FFØ                    /*|Σ_I Δ_I| → AC*/
       TZR(R8) FFØ                    /*Get sign of this term (R8=ø if negative)*/
       SDR(T) FF1 JFL(COMPI,TSTQ)     /*|Σ_I Δ_I| → T, test sign*/
COMPI: CIA(T) FF1                     /*Make Σ_I Δ_I negative if necessary*/
 TSTQ: TZR(R9) FFØ                    /*Get sign of Σ_Q Δ_Q*/
       ILR(R7) FFØ JFL(COMPQ,NADD)    /*|Σ_Q Δ_Q| → AC, test sign*/
COMPQ: CIA(AC) FF1                    /*Make Σ_Q Δ_Q negative if necessary*/
 NADD: ALR(T) FFØ                     /*Σ_Q Δ_Q +Σ_I Δ_I → AC, T*/
       TZR(AC) FFØ KSB STZ            /*Get numerator sign bit in Z-flag*/
       CSR(R9) FFZ JFL(NSHFT, COMPN)  /*Save numerator sign in R9*/
COMPN: CIA(T) FF1                     /*Take abs value of numerator*/
NSHFT: SRA(T) FFØ                     /*Divide numerator by 16 and*/
       SRA(T) FFØ                     /*leave in T*/
       SRA(T) FFØ
       SRA(T) FFØ
```

```
        ILR(R3) FFØ                     /*Denom → AC*/
        CIA(AC) FF1                     /*Form negative divisor*/
        SDR(R8) FF1                     /*Save in R8*/
        CSR(R5) FFØ                     /*Set R5 to all 1's*/
        TZR(R5) KNDLC                   /*Set R5 to negative divide loop count*/
                                        /*(-18)*/
        CLR(R7)                         /*Clear quotient register*/
DLUP:   ILR(R8) FFØ                     /*Neg divisor → AC*/
        ADR(T) FFØ STC INH              /*Get sign of dividend-divisor*/
                                        /*(1 for pos.) & save in C-flag*/
        INR(R5) FF1 STZ JFL(DNEG,DPOS)  /*Increment loop count,*/
                                        /*save its overflow in Z-flag & test sign*/
DPOS:   ALR(T) FFØ JMP(DLS)             /*If pos, dividend-divisor → AC, T*/
DNEG:   ILR(T) FFØ                      /*If neg, dividend → AC*/
 DLS:   ALR(T) FFØ                      /*Shift dividend left 1 bit & store in T*/
        ILR(R7) FFØ                     /*Quotient → AC*/
        ALR(R7) FFC JZF(DLUP,QSGN)      /*2*old quotient + new quotient bit in*/
                                        /*C-flag→ R7, test loop count*/
QSGN:   TZR(R9) FFØ                     /*Get sign of numerator & quotient*/
                                        /*Ø for negative*/
        ILR(R7) FFØ JFL(COMPO,ACCUM)    /*Magnitude quotient → AC; test sign*/
COMPO:  CIA(AC) FF1                     /*Make quotient negative if necessary*/
ACCUM:  INR(R1) FF1                     /*Increment main loop count*/
        ALR(RØ) FFØ JFL(WT2,DONE)       /*Accumulate residue, loop if not the 16th*/
 DONE:  LMI(RØ) FFØ OST JMP(WT1)        /*After 16th residue, output*/
                                        /*sum thru MAR, generate output*/
                                        /*strobe and go wait for next DTI pulse*/
```

The total number of microinstructions in the above program is 103. It is recommended that a 256- or 512-word microprogram memory be used to ease the task of assignment of the code to memory. The worse case number of microinstructions executed in computing and accumulating one of the 16 residues is:

| | |
|---|---|
| 4 multiply loops, 10 passes each, 4 instructions max. per pass | 160 |
| 1 divide loop, 18 passes 7 instructions per pass | 126 |
| Other | 71 |
| TOTAL | 357 microinstructions |

Thus the maximum time required to compute and accumulate one residue is

357 microinstructions x 125ns/microinstruction = 44.6 μs.

This is well under the interval between radar pulses (200 μs) and more than twice as fast as the present ATRAC hardware (~120 μs).

The computation time remains well under 200 μs even if the input word length is increased from 11 to 18 bits, the output word length is increased from 15 to 23 bits, and up to 48 residues are accumulated instead of 16. Specifically, each of the 4 multiply loops requires 7 more passes, adding 11.2 more instructions and 14.0 μs to the above totals. The divide loop requires 9 more passes: 7 due to the increase in output word size, and 2 to maintain output accuracy with respect to truncation error in view of the increased number of residues. Hence the divide routine takes 63 more instructions and 7.9 μs. The division of each residue numerator by 48 instead of 16 would require a few additional instructions. Thus the expected total computation time per residue is expected to be about 67 μs for this projected change. (Note also that the CPE word length must be expanded to 36 bits).

The following table estimates the number of LSI packages required for the implementation of ATRAC with both present and proposed word lengths. Not included in these estimates are random logic for strobe generation, K-field decoding, etc.

| LSI PACKAGE DESCRIPTION | QUANTITY | |
|---|---|---|
| | Present ATRAC (11 bits in, 15 bits out) | Proposed ATRAC (18 bits in, 23 bits out) |
| Intel 3001 MCU | 1 | 1 |
| Intel 3002 CPE 2-bit slice | 11 | 18 |
| Intel 3003 Look-Ahead Carry Generator | 3 | 5 |
| Intel 8212 8-bit latch for pipeline register | 3 | 3 |
| Intel 8212 8-bit latch for data input | 3 | 5 |
| Intel 3604 Bipolar PROM (512 x 8) | 3 | 3 |
| TOTALS | 24 | 35 |

Table 4.2. Parts estimate

## 4.5. Utilizing PLA's in the ATRAC Design

Considering the preliminary investigation of PLA useage as given in section 3.2 (the RMS module) it will be difficult at this point in time to generate the top-down approach. More investigation is required to fully utilize this approach.

As a practical compromise it seems desirable to consider more explicit uses of the PLA within the ATRAC module. With this limited view it will still be possible to meet the following objectives:

(a) gain experience in the application of PLA's,

(b) upgrade the performance of the next generation ATRAC,

(c) reduce the package count, and

(d) improve the design flexibility.

The modifications required in the next ATRAC will not modify the basic computation algorithm but will increase the data widths and increase the number of residues processed per dwell time. The design changes required to increase the data width are fairly obvious and straightforward. The processing of additional residues implies a change in the timing and control.

This section will focus on the changes required in the control unit (Figure 22 , sheet 11 of the ATRAC description). As a practical constraint only FPLA's will be considered. Specifically the Signetic 82S100, 82S101 series will be used. This device is a 28 pin 16 x 8 x 48 field programmable device.

The current cycle time of the controller is 400ns and the PLA delay time 50 μs. The basic block diagram of the modified controller is given in Fig. 4.6.

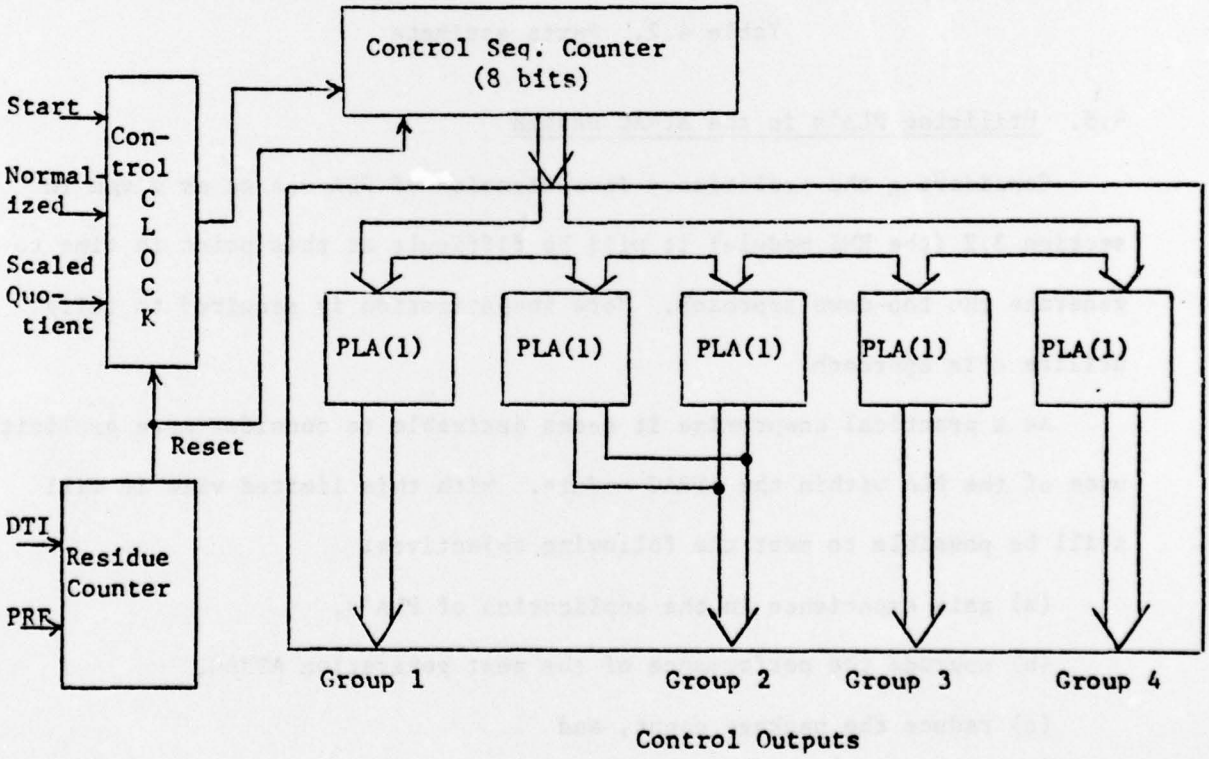



Fig. 4.6. ATRAC Controller

| | | | |
|---|---|---|---|
| Group 1 | CIDR1 | Group 3 | CX1 |
| | CIDR2 | | CY1 |
| | CS1 | | CX2 |
| | CS2 | | CY2 |
| | CS3 | | DIVSTR |
| | M | | DIVSTP |
| | MBS | | DENR |
| | | | CSR1 |
| Group 2 | A | Group 4 | CSR2 |
| | B | | CSR3 |
| | C | | CA |
| | CSM | | CB |
| | AS2 | | INTR |
| | ACC1 | | STOP |
| | ACC2 | | OBRCE |
| | AMC | | |

Table 4.3. PLA Output Groupings

The output terms were grouped to meet the geometric constraints of the PLA. An analysis program was written to derive the PLA product terms implied by the various output group requirements. All groups had fewer than 48 (e.g., group 1 = 37 p terms) except group 1 which required 58. The extra product terms can be gained by parallel devices as shown in Fig. 4.6. The only other modification required is in the residue counter if triple pulse cancelling is no longer used. The execution time of one iteration (nominally 85 μs) is not changed. This time is more than adequate to allow processing of a residue every PRF cycle. The package requirement is reduced from the present 33 to <10. The actual package count will depend upon the number of residues processed.

## 4.6. Comparison of Approaches

The approaches taken boil down to a comparison of the pre ent approach versus a 3000 implementation. This is true since the 8080 is completely inadequate and the PLA approach incomplete. The preliminary results on the use of the 3000 is very encouraging and there seems to be very little doubt that it can meet the system specification. A summary is given in Table 4.4.

Table 4.4. ATRAC Comparison

| Approach | Packages | Execution time |
|---|---|---|
| Conventional logic | 130 | 85 - 120 μs per iteration |
| 8080 8 bit processor | 120 | 1.4 ms per iteration pipelined |
| 3000 bipolar slice | 35 | 67 μs per iteration |
| FPLA | incomplete 3:1 improvement in controller | same as present for controller |

# 5. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

The purpose of this section is to pull togetner the results of the investigation and to make projections and recommendations concerning the application of LSI devices to the MICOM signal processing environment.

## 5.1. Summary

This section will typify the general nature package count, processing throughput and design flexibility of each of three approaches. These three approaches

(a) microprocessors of the Intel 8080 class,

(b) slice oriented microprogrammable microprocessors of the Intel 3000 class, and

(c) programmable PLA's.

represent the present spectrum in LSI devices.

It has become clear that the speed requirements of the MICOM signal processing environment in order to make the use of current LSI devices feasible demands careful system organization. System implementation will involve the use of multiple devices on a module basis.

### 5.1.1. Intel 8080 Class Devices

On a module basis an 8080 implementation will not require significant amounts of buffering packages to achieve the required bus drive capability.

However, the basic requirements for data and control latches, memory, clockery and interfacing require in the neighborhood of 15 packages for a basic 8080 module. There are other manufacturers' parts which could result in lower module package count. A ball park number for module package count still will be in the neighborhood of 10 packs.

The present cycle times of memory and processors limit throughput to the millisecond range if any significant processing is required. The basic operations required in signal processing are not the primitive operations of existing microprocessors. This requires that many cycles be used to affect very simple operations. With signal processing modules time demands in the nano and micro second range this makes the processor out of range by several orders of magnitude.

Speed characteristics may be greatly improved by placing critical operations in faster logic. The tradeoffs of flexibility and package count do not make this particularly attractive.

The ease of design and program modification makes these devices extremely attractive. This is particularly true in the experimental radar signal processing (ground based) area.

5.1.2. Bipolar Slices

The package count of a bipolar slice class implementation is directly associated with the data width requirement. The use of main memory must be eliminated because of the additional timing overhead. The package count as a function of data width n is given in Table 5.1.

Table 5.1. 3000 Module Parts Count

| | |
|---|---|
| MCU (Micro Control Unit) | 1 |
| *ROM (512 x 8) | ~3 |
| *Pipeline Register (8 bit) | ~3 |
| CPE (Processing Elements) | n/2 |
| Carry look ahead | n/8 |
| Input/output latches | n/8 |

*based on a 24 bit microcontrol word.

For 24 bits this comes to about 25 packages.

The throughput for the examples studied was found to be in the microsecond range. That is about 3 μs for the RMS module and approximately 60 μs for ATRAC. This is assuming a basic cycle time of 90ns and using pipeline overlap with carry look ahead.

The flexibility and ease of design modification with this type of device is not as good as with a conventional processor. These devices are fairly difficult to program. It is even more difficult to map the program requirements to the ROM memory. A problem change could easily result in a memory expansion requirement.

5.1.3. PLA

The results of applying PLA to the implementation of a module is much less conclusive than the other two approaches.

The present delay time parameters (50ns) make these devices attractive. The geometric constraints require cascading and it is not clear at this time the ultimate impact that this will have on overall module delay.

The ease of design modification and package count are also unresolved questions. The example problems show sufficient promise that more investigation is warranted. The final answer may involve a mix of PLA's with other LSI devices.

## 5.2. Conclusions

The experience gained by working with the case study designs results in a number of conclusions. These conclusions are categorized below.

System organization

The critical time constraints of the hardware found in the MICOM signal processing environment create the need for careful organization of the system into modules. The modularization defined for previous generation systems has been found to be reasonable in all cases studied.

There were no instances where the use of microprocessors allowed larger modules than previously defined. System implementation will involve the use of multiple processor configurations.

Experience with the 3000 series slice architecture clearly rules out the use of main memory instructions in the implementation of signal processing modules.

There were no cases where an 8080 class device could be utilized due to the speed of the device.

System throughput

The time required for a module to input data, perform a computation, and produce a result (throughput) is very sensitive to the data width and the complexity of the computation required. At this time there is a severe restriction in the throughput possible using 8080 class microprocessors because of their limited data width capability. Both 8080 class and slice oriented microprocessors have operations limited to ADD and SUBTRACT. The complexity of operations in signal processing problems is well beyond this level.

Package Count

LSI devices very obviously improve the density of logic possible. This advantage is diluted at the present time by the speed requirements which create the need for external buffering (FIFO's) and faster, more complex operations (multipliers, etc.).

Other Devices

The programmable logic array (PLA) holds considerable promise due to its logic density and speed. Use of this device is limited by the availability of adequate I/O pins, product terms and its cost (masked vs. field programmable capability). At the present time systematic design methods do not exist to properly take advantage of the power of these devices.

Fast buffer memories (FIFO's, etc.) can provide increased performance in situations where the data is produced in bursts.

Projections

Device performance and cost is changing at a very rapid rate. In addition, the availability of new complex function packages changes daily. In light of this it is important to try to make some projections as to the future applicability of LSI devices to signal processing problems.

Device performance with respect to delay time is improving. However, the basic organization of data processing in a computer limits what can be achieved. The underlying fetch, execute sequence and the notion of stored programs results in a large number of cycles to accomplish a given task. For this reason improvements in the fundamental cycle time do not dramatically improve overall performance. Two specific examples of this exist.

The 8080 instruction set has been implemented in a bipolar slice version (3000 series) and in an ECL machine. The ECL technology represents a bound of the lower limit of delay times which can be achieved. The 8080 instruction execution improvement using ECL was a factor of about 5. The 8080 performance vs. the signal processing throughput demand was 2-3 orders of magnitude apart in the case studies.

Closing the performance gap will depend upon the design of special purpose LSI processors with improved instruction set rather than improved technology. At the present time there is no particular pressure to produce such machines

## 5.3. Recommendations

The following recommendations are made based on the experience reported here and on the foregoing view of the future of technological improvements.

Immediate Goals:

1) Implementation of selected modules using bipolar slice devices. These devices offer the best available solution to processing speed and data width requirements. The next generation ATRAC module is the most attractive candidate.

2) Personnel training in the design methodology of programmable logic design is an essential step in future system development. This has begun and should be continued. There is little doubt that some form of programmed logic will be employed in future signal processing applications. Specific implementation tasks will enhance the training goal.

3) System design and organization studies should be pursued. At the present time it seems likely the necessary operating speeds can only be obtained by systems which utilize some small scale fast logic. System organizations must be defined in such a way as to maximize flexibility, minimize package count and meet performance objectives. This type of study will prepare the way for incorporation of new devices as they become available.

   For example, the PLA is a prime candidate for certain segments of existing designs. The specifics of where and how to utilize these devices requires more consideration.

Long-Range Goals:

1) Identify and specify devices particularly suited to the problems of this environment. The largest gains in terms of the goals of flexible, fast systems requiring few packages can be realized by developing special purpose devices. An example of such a device would be a processing element capable of complex single step operations (multiply, divide, square root, etc.).

2) Development of design tools to facilitate the development of signal processing systems. The concepts and techniques developed in the research laboratory environment can have a large impact on future fielded systems of these ideas can be transported. The development of systematic machine aided design techniques is one way of accomplishing this.

## DISTRIBUTION

| | No. of Copies |
|---|---|
| Defense Documentation Center<br>Cameron Station<br>Alexandria, Virginia 22314 | 12 |
| Commander<br>U.S. Army Materiel Development and Readiness Command<br>ATTN: DRCCG<br>DRCRD<br>5001 Eisenhower Avenue<br>Alexandria, Virginia 22304 | <br><br>1<br>1 |
| Commander<br>Ballistic Missile Defense Systems Command<br>ATTN: BMDSC-HR, Mr. Stevenson<br>P.O. Box 1500<br>Huntsville, Alabama 35807 | 1 |
| Director<br>Ballistic Missile Defense Advanced Technology Center<br>ATTN: ATC-R, Mr. Carlson<br>P.O. Box 1500<br>Huntsville, Alabama 35807 | 2 |
| Commander<br>US Army Electronics Command<br>ATTN: DRSEL, Mr. Fishbien<br>DRCPM-MALR<br>Fort Monmouth, New Jersey | <br>1<br>1 |
| DRCPM-MDE, Mr. Wood | 1 |
| DRCPM-HAE, Mr. Ams | 1 |
| DRCPM-CFE, Mr. David | 1 |
| DRCPM-SHO, Mr. Bishop | 1 |
| DRSMI-R, Dr. McDaniel | 1 |
| -R, Mr. Fagan | 1 |
| -R, Dr. Kobler | 1 |
| -RE, Mr. Lindberg | 1 |
| -RE, Mr. Pittman | 1 |
| -REO, Mr. Currie | 1 |
| -RER, Mr. Cash | 1 |
| -REG, Dr. Burlage | 50 |
| -RE, Record Set | 1 |
| -RG, Mr. Huff | 1 |
| -RD, Dr. McCorkle | 1 |
| -RBL | 5 |
| -RP, Dr. Jackson | 1 |